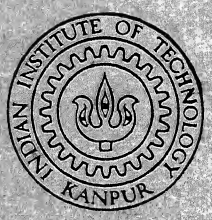
A SIMULATOR FOR A SYSTOLIC ARRAY SIGNAL PROCESSOR (SASP) BASED ON ADSP - 14XX AND 32XX CHIP SET

by

SANJAY A. WANDHEKAR



DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

February, 1990

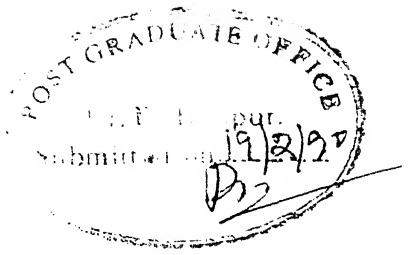
A SIMULATOR FOR A SYSTOLIC ARRAY SIGNAL PROCESSOR (SASP) BASED ON ADSP - 14XX AND 32XX CHIP SET

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by
SANJAY A. WANDHEKAR

to the
DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
February, 1990



C E R T I F I C A T E

It is certified that the work contained in the thesis entitled " **A Simulator for a Systolic Array Signal Processor (SASP) based on ADSP - 14XX and 32XX chip set** " by Mr. Sanjay Annasaheb Wandhekar , has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Dr. A. Joshi)

Professor

Department of Electrical Engineering,

Indian Institute of Technology ,

Kanpur 208016.

February , 1990.

Acknowledgements

It gives me great pleasure in expressing my sense of great gratitude to all those who were instrumental in completing this thesis.

My biggest debt of gratitude goes to Dr. A. Joshi , who gave me constant encouragement and valuable guidance.

I wish to express my special appreciation to Dr. A. Mahanta for suggesting the thesis topic , for his continued support and encouragement throughout the work.

The discussions with Subramanian helped me a lot , and I am indebted to him. I am grateful to Nitin Gogate , Ganesan , Shera and Ranade for their timely help and cheerful company. I would like to express my heartful thanks to Kasi , Ravindra Babu, Jasuja, Maniyar, Diagavane and Bhandarkar for the cheerful company during my leisure time.

CONTENTS

1. INTRODUCTION

The increasing demands of speed and performance in modern signal and image processing applications necessitate a revolutionary super-computing technology. Sequential systems will be inadequate for future real-time processing systems, and the tremendous computational capability of the array processing will become a necessity. In most real-time digital signal processing applications, general purpose parallel computers cannot offer enough processing speed due to severe system overheads. Therefore, special-purpose array processors have become the only appealing alternative.

1.1 ARCHITECTURES :

Parallel computers can be divided into three architectural configurations : [Hwang84]

-pipeline computers or vector processors.

-Multiprocessor systems.

-Array processors.

A pipeline computer performs overlapped computations to exploit 'temporal parallelism'. A multiprocessor system achieves 'asynchronous parallelism' through a set of interactive processors with shared resources (memories, etc.). An array processor uses multiple synchronized processors to achieve 'spatial parallelism'. The first two classes belong to the general-purpose computer domain. The development of these systems requires a complicated design of control units and optimized schemes for the allocation of machine resources.

The last class of computers offers a promising solution to meet real-time processing requirements. In particular, locally interconnected computing networks, such as systolic and wavefront arrays, are well suited to efficiently implement a major class of signal processing algorithms, due to their massive parallelism and regular data flow [HTkung82]. Therefore, we will focus on, the systolic architectures.

1.2 SYSTOLIC ARCHITECTURES:

In a systolic system, data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory, much as blood circulates to and from the heart [HTKung82]. The array can be linear rectangular, or hexagonal to make use of higher degrees of parallelism.

Computational tasks can be classified into two families : Compute-bound computations and I/O-bound computations [HTKung82]. In a computation, if the total number of operations is larger than the total number of input and output operations, then the computation is compute-bound, otherwise it is I/O-bound. For example, ordinary matrix multiplication is compute-bound, whereas adding two matrices is I/O-bound. Speeding up the I/O-bound computations requires an increase in memory bandwidth, which is limited by the current technologies. Speeding up a compute-bound computation, however, can be accomplished using systolic arrays. The basic configuration of a systolic array is as shown in fig. 1.1. By replacing a single processor by a 1-D or 2-D array of processing elements (PE), a higher computation throughput can be achieved without increasing memory bandwidth.

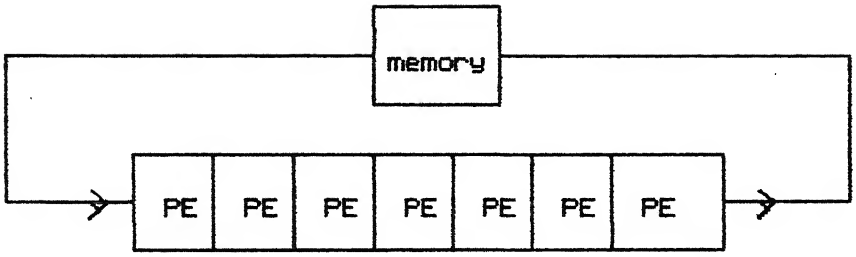


Fig. 1.1 Basic configuration of Systolic Arrays.

1.2.1 Definition of Systolic Arrays :

A systolic array is a computing network possessing the following features : [HTKung82], [SYKung87].

-Synchrony : The data are rhythmically computed (timed by a global clock) and passed through the network.

-Modularity and regularity : The array consists of modular processing units with homogeneous interconnections. Moreover, the computing network may be extended indefinitely.

-Spatial locality and temporal locality : The array manifests a locally - communicative interconnection structure, i.e. spatial locality. There is at least one unit-time delay allocated so that signal transactions from one node to the next node can be completed, i.e. temporal locality.

-Pipelinability : (i.e., O(M) execution-time speedup). The array exhibits a linear rate pipelinability, i.e., It should achieve an O(M) speedup, in terms of processing rate, where M is the number of processing elements (PEs).

1.2.2 Why Systolic Architectures ?

The major factors because of which, systolic arrays are useful for special-purpose processing architectures are : simple and regular design, concurrency and communication, and balancing computation with I/O [HTKung82].

Simple and Regular Design : By using a regular and simple design, great saving in design cost can be achieved. Furthermore, simple and regular systems are likely to be modular and therefore adjustable to various performance goals.

Concurrency and Communication : There are essentially two ways to build a fast computer system. One is to use fast components, and other is to use concurrency. Since the technological trend clearly indicates a diminishing growth rate for component speeds, for major improvement in the speed, concurrent use of many processing element is essential. When large number of processors work together, communication becomes significant. In VLSI technology, routing costs dominate the power, time, and area required to implement computation; therefore, regular and local communication in systolic arrays is advantageous.

Balancing Computation with I/O : The ultimate performance goal of an array processor system is a computation rate that balances the available I/O bandwidth with the host. With the relatively low bandwidth of current I/O devices, to achieve a faster computation rate, it is necessary to perform multiple computation per I/O access. However, the repetitive use of a data item requires it to be stored inside the system for sufficient length of time. In other words, the I/O problem influences not only the required I/O bandwidth but also the required internal memory. The question then is how to arrange a computation together with an appropriate memory structure and I/O bandwidth, so that computation time is balanced with I/O time.

The I/O problem becomes especially severe when a large computation is

performed on a small array. In this case, the computation must be decomposed (partitioning problem). In practice, this is often the case, and therefore, questions such as how a computation can be decomposed to minimize I/O, and how buffer memory can be arranged to minimize I/O are critical to the practical design of an array processor system.

A solution to above challenges is the systolic array processing. A systolic system consists of a set of interconnected simple cells. Information in a systolic system flows between cells in a pipelined fashion, and communication with the outside world occurs only at the "boundary cells". Thus computation rate of the system can be balanced with available I/O bandwidth with the host.

1.3 OBJECTIVE AND SCOPE OF THE CURRENT WORK :

In this thesis an attempt has been made to develop a simulator for a linear systolic array processor for developing and executing a class of signal processing algorithms. The objectives of the current work are described below.

1) To define a systolic array architecture, which will work as an attached processor to an external host.

2) To develop a simulator for this architecture using ADSP 14XX and 32XX chip set.

3) To develop a generalized, redefinable microassembler (meta-assembler) , which is used for developing programs for the simulator.

4) To test the simulator and meta-assembler through execution of some simple systolic algorithms.

1.4 ORGANIZATION OF THE REPORT :

Chapter 2 presents an overall view of the systolic array signal processor (SASP) , which has been simulated. We follow the array configuration proposed by Nemawarkar [NEM88]. A simplified version of the SASP system hardware has been developed in earlier thesis and described in [Usman89] and [SAM89]. Several additional details have been incorporated in the simulator described in this thesis , though some of these features are not included in the present hardware development , under progress [Subramanian90], [Shera90].

Chapter 3 details the design and implementation aspects of the SASP. The chapter discusses the design of processing element (cell) and the interface unit (IFU).

Chapter 4 gives the simulator details for the SASP system. The methodology adopted for the simulation is given.

Chapter 5 discusses the algorithms for matrix multiplication and convolution on the SASP system. Verification of the meta-assembler and simulator is done using these programs.

Chapter 6 describes the generalized, redefinable microassembler (meta-assembler) , which can be used for any microprogrammed architecture. The facilities provided in the meta-assemler and its operation are also given.

Chapter 7 gives the conclusions and lists few suggestions for future work.

The manual for the meta-assembler , manual for the Simulator , the definition files for the microcode of the interface unit and the cell and the manual for ADSP-1401(sequencer), ADSP-1410 (address generator) , ADSP-3210 (Multiplier) and ADSP-3220 (ALU) chips are given in the Appendices A, B, C and D respectively.

2. SYSTEM OVERVIEW

2.1 INTRODUCTION:-

In this chapter, the Systolic Array Signal Processor (SASP) has been introduced. It is a systolic array computer of linearly connected cells, each of which is a microprogrammable processor capable of performing floating-point operations. It is designed for computation-intensive applications.

The architecture is similar to the WARP computer developed at Carnegie Mellon [Anna87]. The system has been simulated on a PC , as well as on a HP-9000 system and the working is tested by developing some algorithms. Some utilities (e.g. Assembler) have been used in the development of system hardware.

In this chapter the architecture of SASP system and its main features are discussed.

2.2 ARCHITECTURE:

The SASP system is attached to a general purpose host PC/XT through a parallel interface. The system , thus consists of three major parts - the Host, interface unit (IFU) and the processor array. fig. 2.1

1) **HOST** :- The host supplies data to the array and receives the results from the array. It programs the array to execute algorithms. In addition, it executes those parts of algorithms, which cannot be mapped onto the array.

2) **Interface Unit (IFU)** :- The interface unit handles the input/output operation between the array and the host, and it generates addresses (Addr) and control signals for the processor array. It also regulates the flow of the intermediate results through the processor array.

3) **Processor-Array** :- The main computing power of the SASP is from the

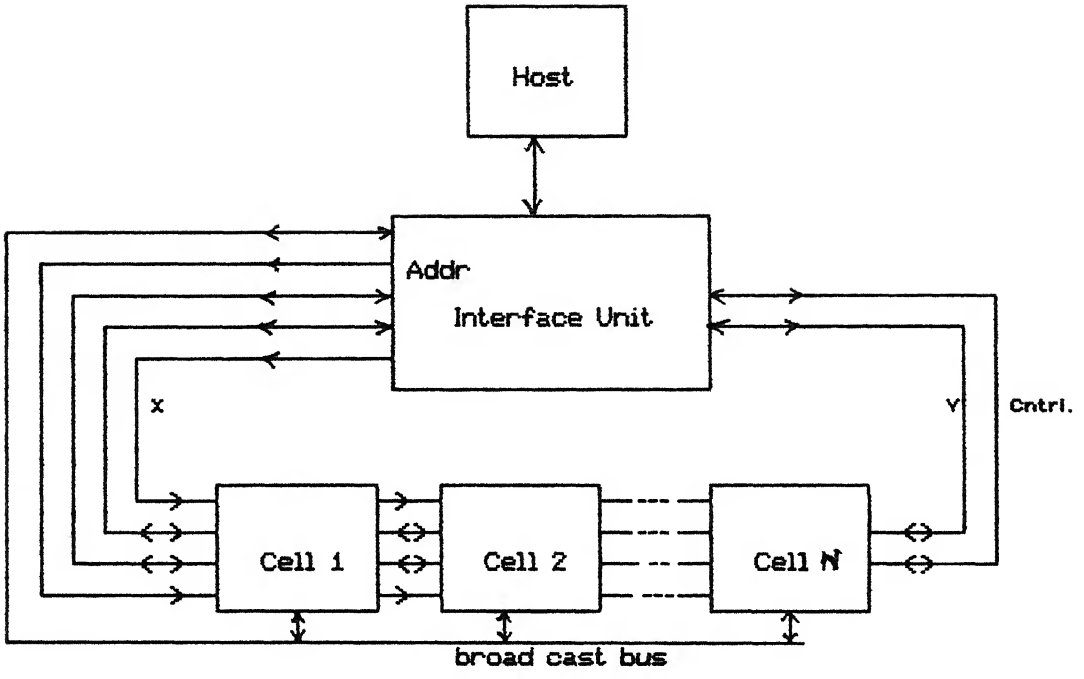


Fig. 2.1 System Overview.

processor array. The array consists of identical cells (processors). Each cell is a programmable horizontal microengine, with its own sequencer, address generator, data memory and program memory. The microcode for the cell can be downloaded by the host through the broadcast bus (BC-bus).

The cells are connected through inter-cell communication channels (X, Y and Addr). Each cell can communicate with its neighbours (left and right). The 32 bit data flows through the array on X and Y channels. The addresses for the local memories of the cell, generated by IFU, propagate down the Addr channel. Moreover, the cell has a local address generator, which helps in efficient loop realizations. The direction of Y channel can be reconfigured at any time, through microprogram bits. This feature can be used, for example, in algorithms that require accumulated results in the last cell to be sent back to the other cells, or require local exchange of data between adjacent cells.

2.3 INTERCELL COMMUNICATION:

In the architecture of the SASP machine the global communication is only through the broadcast bus (BC-bus). The BC-bus is used only for the loading/reading of the microprogram and the data. But during the execution of algorithms only local communication channels are used. A Queue is associated with each channel (XQ ,YQ, and AddrQ) and is placed in the data path of the input cell. Use of queues greatly enhances the intercell bandwidth.

The 'flow control' for the communication channel described below, has been assumed to be implemented in the hardware. When a cell tries to read from an empty queue, it is blocked (i.e. Cycles are skipped) until data item arrives. Similarly, when a cell tries to write into a queue of a neighbouring cell when it is full , the writing cell is blocked until data is removed from the full queue. The

blocking of a cell is transparent to the program. Only the cell that tries to read from an empty queue or to deposit a data item into a full queue is blocked. All other cells in the array continue to operate normally. The data queues of a blocked cell are still able to accept input; otherwise, a cell blocked on an empty queue will never become unblocked.

The channels are described below:

The X channel is a 32-bit wide data path and is unidirectional. It starts from IFU and ends on the last cell. The data on which computation is to be done is transmitted over this channel and it ripples through the cells without being modified.

The Y channel is also 32-bit wide. It is bidirectional and its direction can be statically reconfigured by the algorithm to be executed. This channel forms a closed loop, starting from IFU, running through the processor array and ending at the IFU. Intermediate or final results travel on this channel. At the end of each pass through the processor array, Y channel terminates into YA and YB memories in the Interface unit. Each cell can communicate with both of its neighbours through this channel.

The **Addr** channel provides addresses for local data memories in the cells. Address is generated in the interface unit and transferred along with data on the X-channel.

The **Cntrl** channel contains control signals to read from or write into queues and the status of the queues of the neighbouring cells.

Signals coming to the cell n from cell n+1 are as follows.

Xqfull To indicate X queue of cell n+1 is full.

Yqfull To indicate Y queue of cell n+1 is full.

Addrqfull To indicate Addr queue of cell n+1 is full.

wryq To write data into Y queue of cell n.

Signals going from cell n to cell n+1 are as follows.

wrxq* To write data into X queue of cell n+1.

wryq* To write data into Y queue of cell n+1.

wraddrq* To write address into Addr queue of cell n+1.

yqfull To indicate Y queue of cell n is full.

The signal 'wryq' coming from the cell n+1 is considered only when Y bus direction is reversed.

The IFU is considered as cell 0.

For cell N signals going to the IFU are

wryq* To write into YB memory in the IFU.

wrya* To write into YA memory in the IFU.

2.4 BROADCAST BUS (BC-bus):

The broadcast bus is used by the host to load data and microprograms into each cell. Its signals are as follows.

1. Data lines - For loading/reading microcode and data memory of each cell.

2. Cell address - The addressed cell communicates with the host through data lines.

3. Reset - System reset.

4. Y bus direction - This signal originates from the microcode memory of IFU. It decides the Y-bus direction through the array.

5. Read, write and handshake signals from/to host interface for reading and

writing into the data memories and the microprogram memories of the array cells.

6. Flg - Flag input to all the sequencers in the system. It can be used to indicate start of execution or end of execution of a program. Any cell can raise this signal.

7. Wcs - Assertion of the signal forces the sequencer (ADSP-1401) in a cell to execute WCS instruction. This instruction is used for loading the microcode in microcode memory.

8. CLK - This is global clock broadcasted over the BC bus to be used by all the cells.

2.5 THE CELL UNIT :

The block diagram of a cell is shown in fig. 2.2. Each cell has its own program sequencer and has the data path as shown in figure. The following are the major features of the cell unit data path:

ALU and Mpy: The multiplier and ALU are implemented with commercial 32-bit floating-point multiplier ADSP-3210 and floating-point ALU ADSP-3220 chips, respectively. These chips use pipeline mode to achieve the maximum possible throughput. That is, a chip starts a new 32-bit arithmetic operation every cycle, although the result of an operation will not emerge from the chip's output port until few cycles after the operation starts. Thus the processor array supports pipelining at both the array and the cell levels. These two levels of pipelining greatly enhance the system throughput.

Reg-file: It contains 128 32-bit wide registers accessible from any of the 5 ports. Two ports are input ports, two are output ports and one port is bidirectional. Register file is implemented using ADSP-3128 chips.

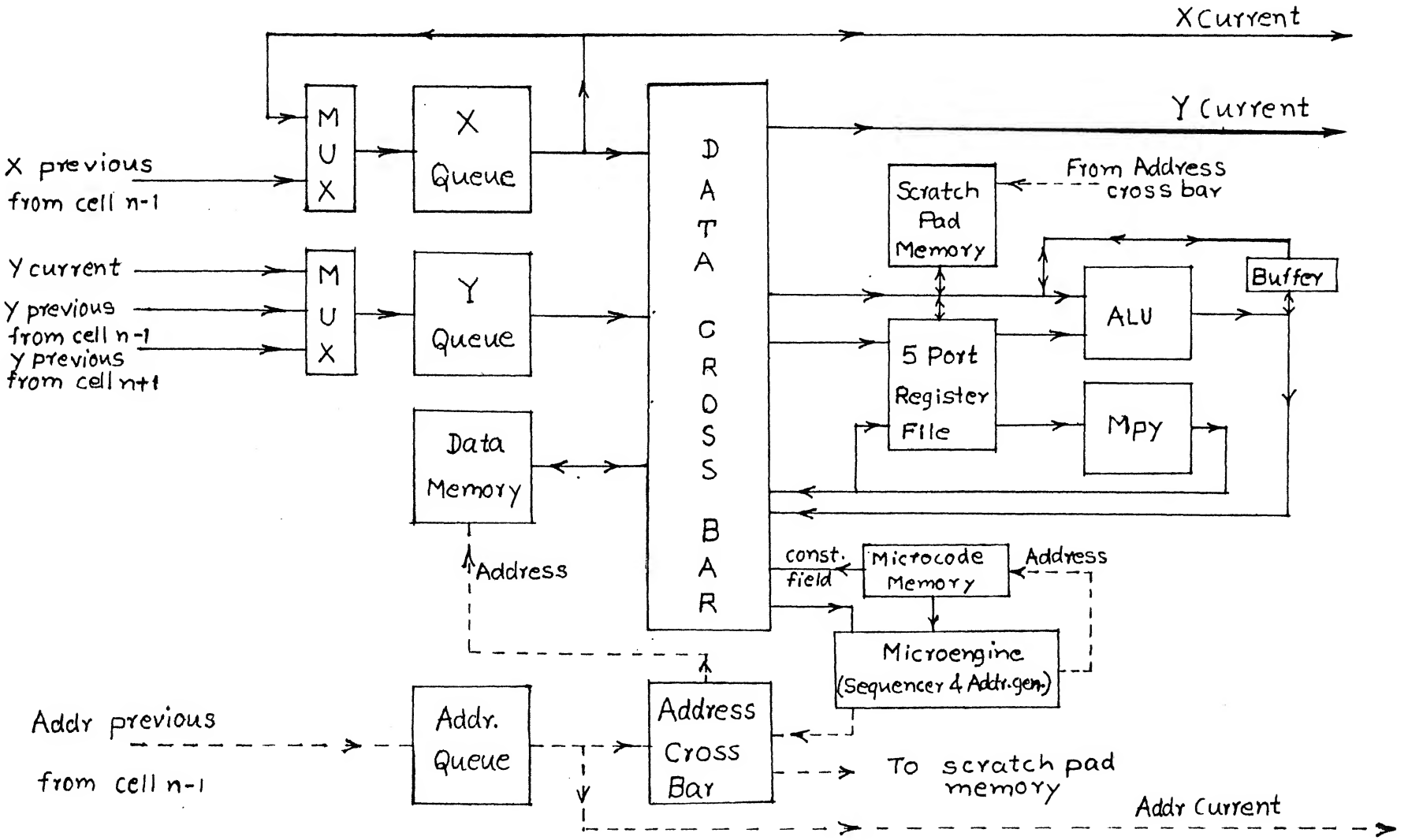


Fig. 2.2 Cell data path

X Queue, Y queue, and Addr Queue : These queues are provided mainly to ensure that X, Y and Addr stream are properly synchronized, as required by systolic algorithms. The queues are implemented using CY7C40 (512 X 9 FIFO) chips.

Data memory: Having a memory at each cell for buffering data, implementing look-up tables , or storing intermediate results is essential for reducing the I/O bandwidth requirement of the cells. Also by using its local memory to store temporary data, a cell can be multiplexed to implement the functions of multiple cells in systolic array design. As a result, for example SASP can implement algorithms designed for two dimensional systolic arrays or one dimensional arrays that have more cells than the one dimensional array of the machine.

Cross bar : The ALU, Mpy, Register files and I/O ports of the SASP cell are linked by a crossbar. The crossbar can be reconfigured every cycle under the control of microcode to allow a read port to get data from any of the six write ports. The cross bar is implemented using multiplexers.

Input multiplexers : These are used to implement computations using the wraparound or bidirectional data flow mode. In the wraparound mode the outputs of the cell is fed back to its inputs, hence wrapping around the cell. This mode multiplexes the use of one cell to implement the function of several. (The same effect can also be achieved through the use of other resources, such as the data memory.) This increases the virtual size of the array for problems requiring larger array size. In the bidirectional data flow mode the Y input of each cell can take values from the Y output of the next cell, that is, the cell to the right, this feature allows the SASP array to implement linear systolic array with bidirectional data flows.

The detailed description of the system is given in chapter 3.

3. SYSTEM DESIGN

In this chapter the details of the system design are described. The functions of each major unit and the overall block diagram has already been introduced in chapter 2.

3.1 *SASP interface Unit (IFU)* :

Fig. 3.1 shows the block diagram of the IFU. The interface to host enables the host to access different parts of the IFU as well as the cells. A list of functions done by the interface unit is given below.

i) To supply X and Y data to the array at required rate.

ii) To route data according to the configuration of the array (Forward or reverse).

iii) Receiving intermediate results and looping them back into the array.

iv) Receiving and storing output results.

v) Generation of address for address bus.

vi) Acting as an interface between array and the host.

Each of these blocks shown in figure 3.1 is described below.

3.1.1 Interface to Host :

This unit contains buffers, decoders and control lines coming from the host. The buffers buffer the address , data , and control lines from the host. The decoder selects proper blocks for writing or reading of data. The control lines

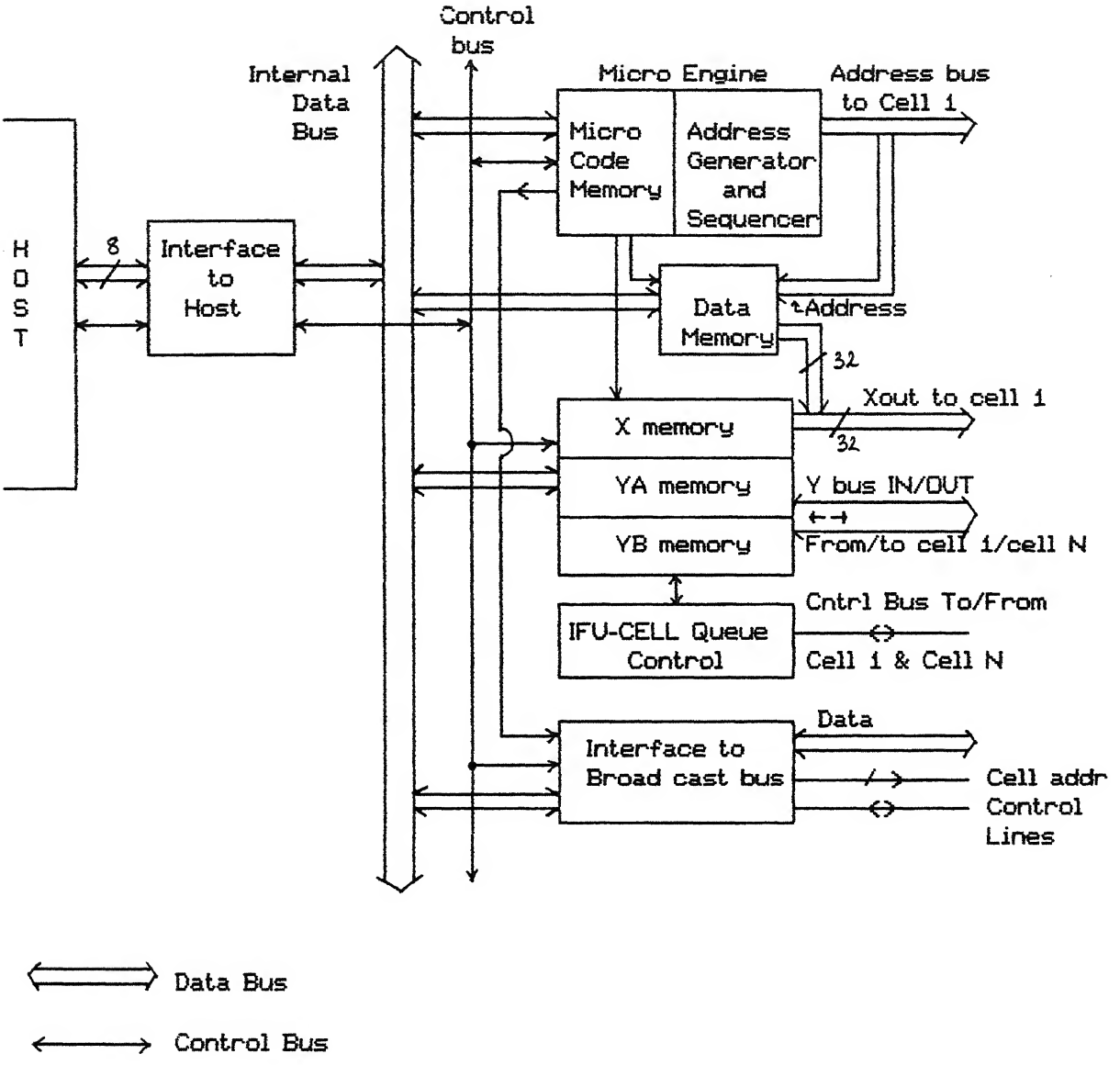


Fig. 3.1 Interface Unit Block Diagram.

are ready, read and write signals.

Before the start of execution by the array the host dumps the data onto the RAMS of the interface unit and onto the data RAMS of each cell over the broadcast bus as described in the section 2.4 (fig 2.1). The code is also dumped onto the microcode RAMS of IFU and the cells , in the same way.

The interface unit takes over control of all the blocks after the host has finished loading of microcode and data, and supervises all kinds of routing and exchange of data during execution of algorithm. After the execution is over, it generates an interrupt , asking the host to take necessary action. Then the host may read the results and may process the received results , further.

3.1.2 Microengine and Address generator unit :

The control unit of the interface unit has been implemented as a programmable microengine. It is a standard practice to use microprocessors to control the data flow and to perform computations in a 'smart' circuit. But the inherent sequential nature of microprocessor operation prevents its use in high throughput machines, where more functional parallelism is required. This parallelism can be achieved by using microcode approach. The main difference between microprocessor circuits and microcode circuits is that the functional units integrated in a microprocessor are spread out as separate building blocks in a microcoded circuit , so that they can operate simultaneously and independently.

In order to coordinate the independently operating devices , these functional blocks are operated in synchronism with a common system clock. Control instructions/signals for each device are put together in a central microcode memory and a microinstruction location is accessed in every system clock cycle.

For such a system , it is necessary to have a sophisticated program flow, that accommodates nested loops, subroutines , interrupts, etc. Such demanding sequencing tasks are met by using a Program Sequencer ADSP-1401. Like other functional units, the sequencer also gets its instructions from the microcode memory and generates address for the next microinstruction depending on the current instruction.

The microcode details for the microengine of the IFU are given in the appendix C in the form of a definition file , which is input to the definition phase of the meta-assembler , which in turn generates a compact definition file. This compact definition file is then used by the simulator and the assembly phase of the meta-assembler to run and to assemble the programs for the IFU.

PROGRAM SEQUENCER:

During each microinstruction , the ADSP-1401 monitors the conditions and instruction to determine the next microprogram address. This address can come from one of several sources : the internal stack , the jump address space in the internal RAM, the data port , the interrupt vectors, or the program counter. The detailed description of the chip is given in the appendix D.

The external flag input to sequencer chip may be used to control conditional instructions. Two instructions make explicit use of FLAG as their condition (JPCOF and JPCNF), while others employ a conditional mode selection (UNCONDITIONAL, NOTFLAG, FLAG or SIGN) to be specified as part of their opcode. Here the FLAG input has the number of sources,. One source can be selected at a time through the control lines to the multiplexer from the microcode .[fig.3.2]. For the IFU there are 4 sources , 'end_op' -indicating end of the execution of an algorithm , 'flg' -can be asserted by any cell under exceptional conditions , 'cmpz' -

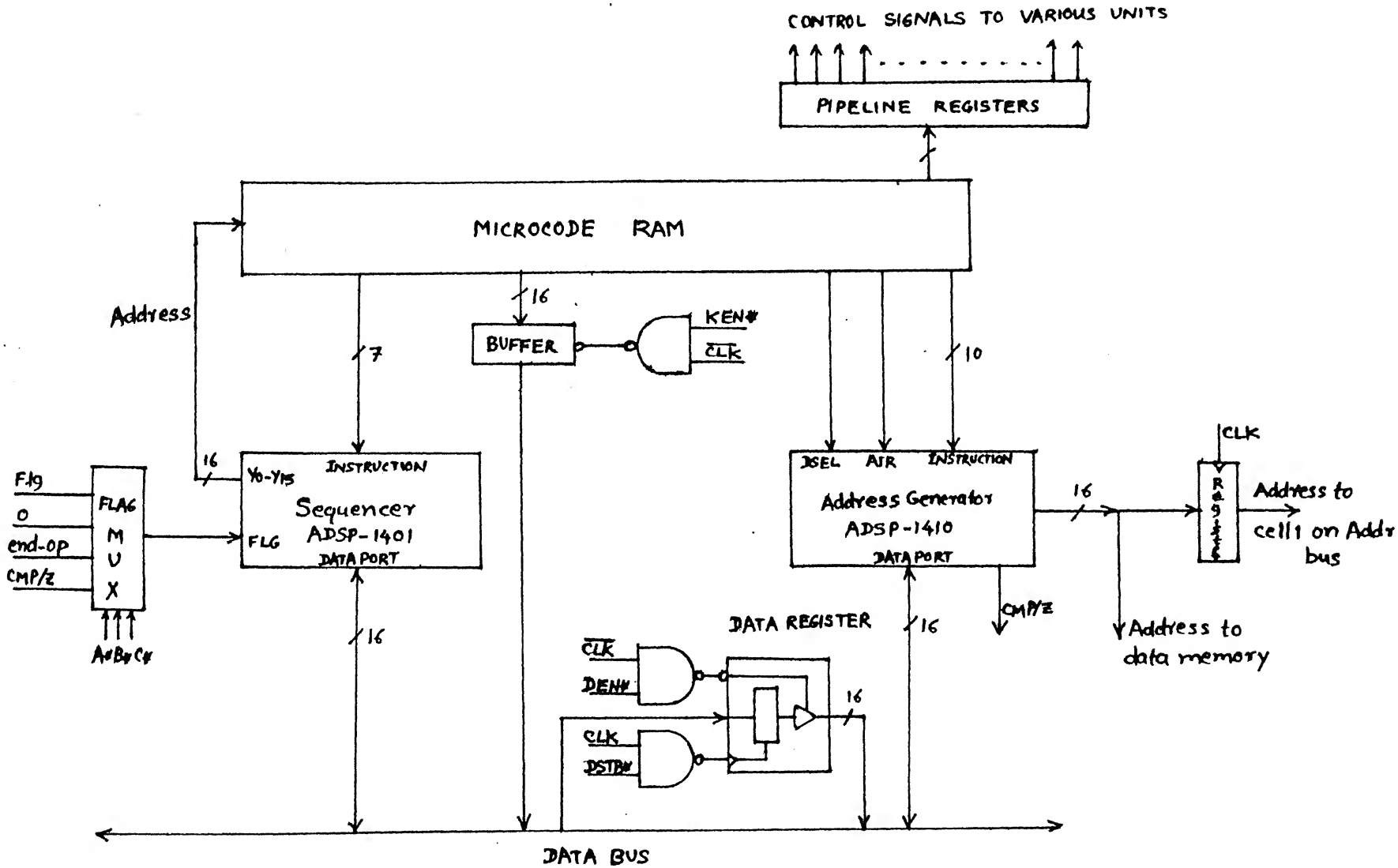


Fig. 3.2 Microengine and Address Generation Unit For the IFU.

generated by address generator and '0' -no flag input. No flag input is the default choice.

Note: In all figures, Signals , whose names are ending with '*', originate from the microcode memory.

ADDRESS GENERATOR :

The interface unit (IFU) has to generate the address on the addr-bus which flows systolically from cell to cell. It also has to address the data memory in the IFU. The ADSP-1410 address generator has been used for flexible address generation. This device rapidly generates the data memory addresses required by routines such as digital filters , FFTs , matrix multiplications and DMAs. Circular buffers and modulo addressing for data memories can be implemented without overhead. In a single instruction the device can :

- Output a 16-bit memory address;
- modify this memory address ; and;
- detect when the address value has moved to or beyond a pre-set boundary and conditionally loop back to the top of a circular buffer.

The details of the chip are given in the appendix D.

3.1.2.1 Data transfer between sequencer and address generator:

The data transfer between sequencer and address generator can be used for saving the 1410 registers on the 1401's subroutine stack during a context switch or subroutine call. In addition, it allows use of the 1410 as an ALU for program addresses. For instance , if this system were performing a FFT, the 1401 would need the shifting function of the 1410's ALU for calculating the number of

butterflies per group and the number of groups per stage. Both are used by 1401 for counting loops in the FFT programs.

The output and input arrangement of ADSP-1401 and ADSP-1410 permits data to be output during clock HIGH , while inputting of data is performed in clock LOW phase, thus allowing reading and writing of data in single clock cycle.

The circuit in fig. 3.2 allows following data transfers in a single clock cycle.

1) constant field to 1401 (ken* =1, den* =dstb* =0).

2) constant field to 1410 (ken* =1, den* =dstb* =0).

3) 1401 <--> 1410 (ken* =0, den* =dstb* =1).

3.1.3 MEMORY STRUCTURE :

The interface unit consists of four kinds of memories. The X, YA and YB memories are accessible only sequentially. The fourth memory is data RAM. X memory is the place for data on which desired computations are to be performed by the SASP array. The X memory can also be written from the data memory , when the data is too large to fit into the X memory. YB memory is used for intermediate results. YA can be used for final or initial results. Details of these memories are as follows.

a) X memory [fig.3.3]:

The host can read and write into the X memory. The IFU can read it to transfer X data to cell 1 over the X-bus. and can write it from the data memory, whenever the previous data loaded is over. Since memory addressing is through a counter, the memory can be accessed only sequentially. Simultaneous read and

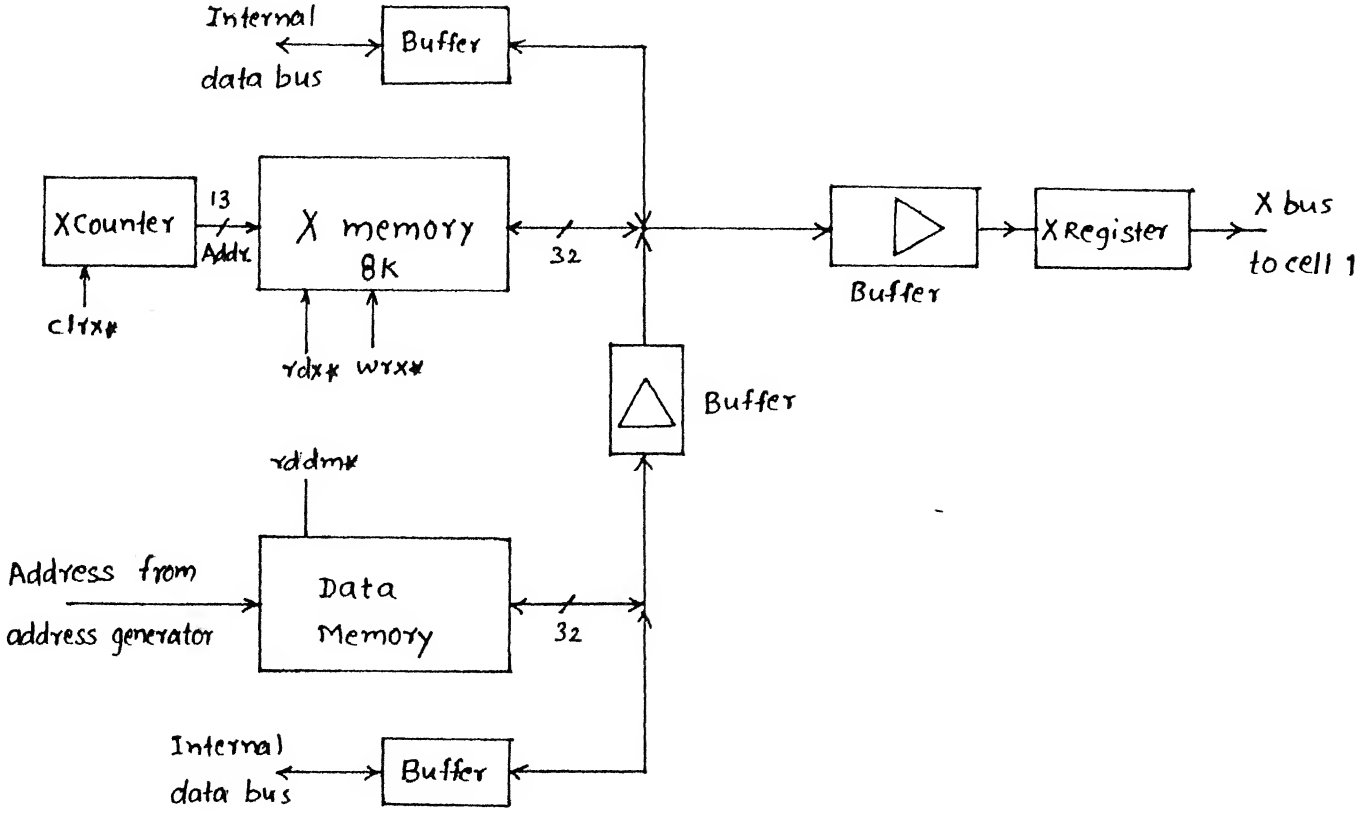


Fig. 3.3 X memory and Data memory for the IFU.

write is not possible. The signals rdx* and wrx* are from the microcode memory. The read/write signals from the host are not shown. The signal clrx* clears the counter to zero.

The signal rdx* strobes the X data into X register. To send/or to write this data into X queue of cell i IFU asserts (through microcode) wrxq* , which is valid if the X queue of cell i is not full , and then the signal acts as write signal for the X queue of cell i. Thus the X register data is written into the X queue of cell i.

b) YA memory [fig. 3.4]:

It can be read and written by the host. The IFU reads the YA memory to transfer YA data (may be initial results) over the Y bus through the SASP array. The cell N or cell 1 (depending on the Y bus direction) can write YA memory.(Final results.). Since here also the addressing is through a counter, memory can be accessed only sequentially. Only read or write can be done at a time. Y bus , which goes through the SASP array is shared by YB memory as well. Output from the YA is strobed into Y register through rdya* . To send the data written into Y register to cell 1/cell N , the IFU microengine asserts wryq* , which is valid if the Y queue of cell 1/cell N (depending on the Y bus direction) is not full, and then acts as write signal for Y queue of cell 1/cell N. Thus the Y register data is written into the Y queue of cell 1/cell N.

c) YB memory [Fig. 3.4]:

This is used for storage of intermediate results and can be accessed by the IFU and cell 1 or cell N (depending on the Y bus direction.) at the same time. The IFU can read it to transfer data onto the Y bus and cell N/cell 1 can write into it.

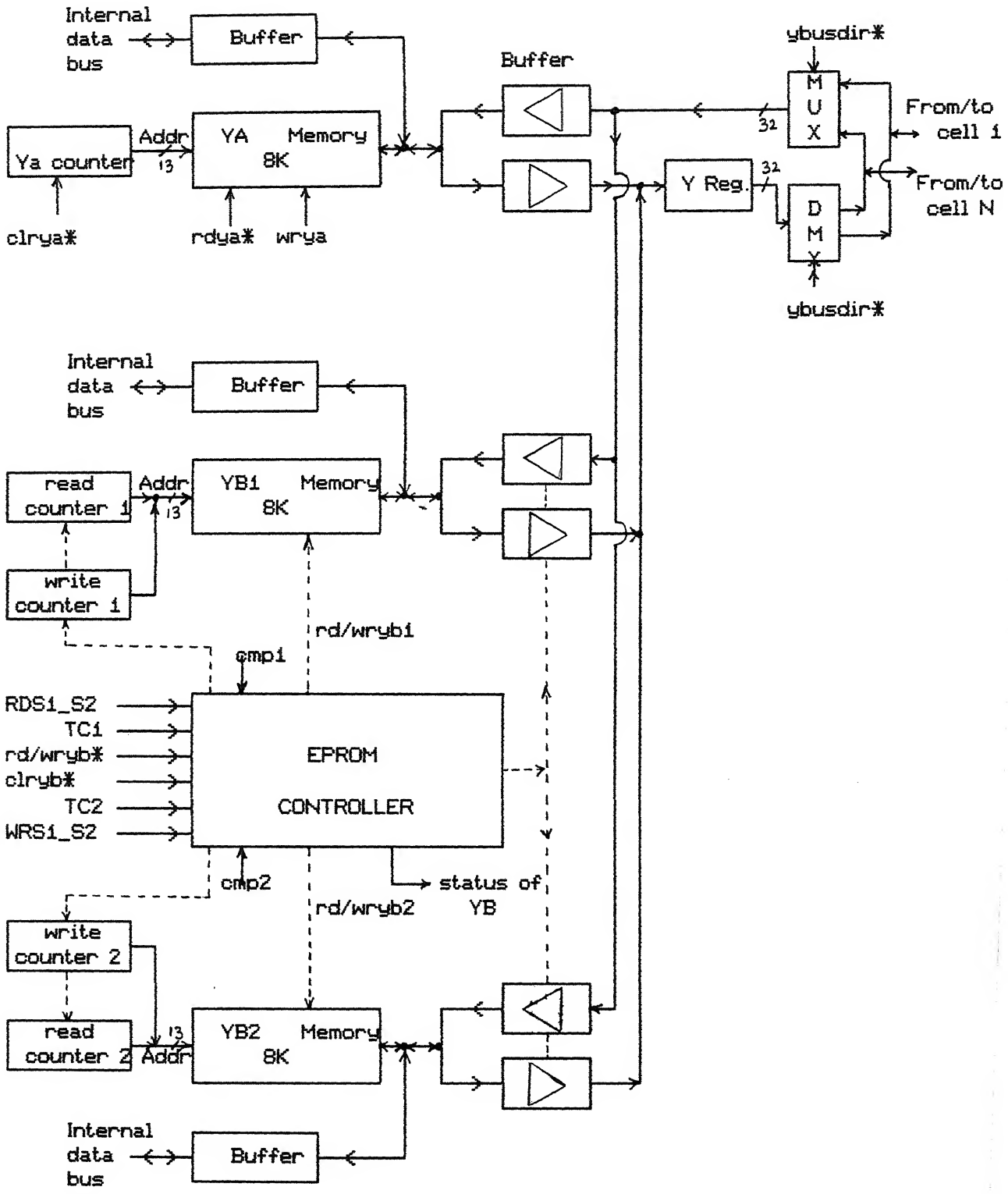
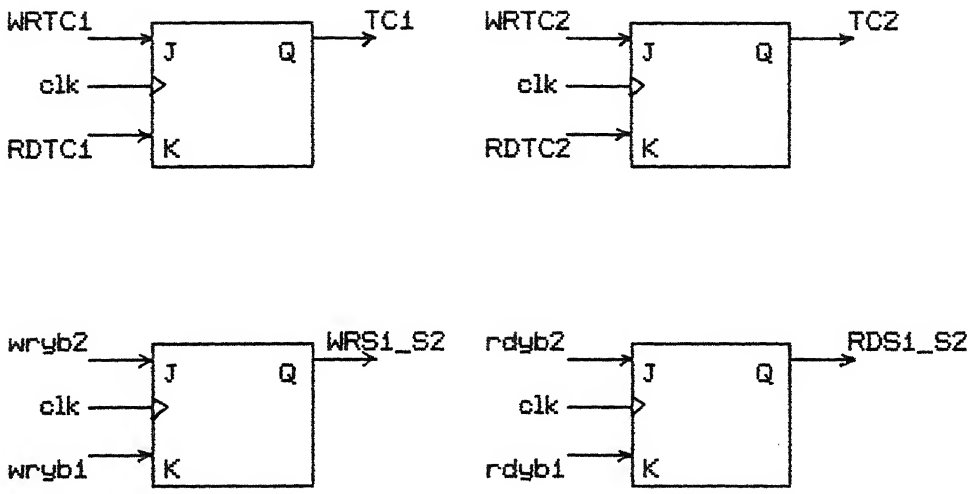


Fig. 3.4 a YA and YB Memories for the IFU.



WRTC1 -Terminal count for write counter 1.

WRTC2 -Terminal count for write counter 2.

RDTC1 -Terminal count for read counter 1.

RDTC2 -Terminal count for read counter 2.

Fig. 3.4 b Signal inputs to the EPROM controller of YB memory.

The simultaneous read and write operation is achieved by using two RAMS, interleaved dynamically. That is, when first RAM was written and read in previous accesses , on simultaneous read and write request , writing is started into the second RAM (YB2) and the first RAM is read at the same time. When the data written into the first RAM (YB1) gets exausted , the IFU microengine starts reading from YB2 and writing continues in the YB2. And writing switches to YB1 if both read and write occur simultaneously.

Writing and reading of YB is controlled using EPROM , flip-flops and gates as shown in fig.3.4.

Two YB RAMS have their own addressing counters, one for read and another for write operation. Reading is allowed when data is already written in the memory. If the IFU tries to read an empty YB memory , the clock of IFU's sequencer is skipped and this skip is lifted only when memory gets a new data item. This skipping elongates read YB cycle and microengine effectively waits for data YB to arrive from cell N/cell 1.

Some signals are explained below to show how simultaneous reading and writing is achieved.

WRS1_S2 and RDS1_S2 are two signals (fig 3.4b), which are LOW when the previous write (read) was into (from) YB1, otherwise are HIGH.

TC1 and TC2 : (Terminal counts for YB1 and YB2) When the terminal count for the write counter of YB1 (YB2) is reached TC1 (TC2) is set HIGH and it is cleared to LOW when the terminal count for the read counter of YB1 (YB2) is reached.

CMP1 and CMP2 : When both the read counter and write counter for YB1 (YB2) are pointing to the same location CMP1 (CMP2) is made HIGH. A comparator is used for the comparison of read and write counter values.

YB1full : This is asserted when YB1 memory is full. It is ensured when terminal

count TC1 is high and both read and write counters are equal. i.e.

$$Ybifull = cmpi \cdot TC1 \qquad -(1)$$

YB2full : Similarly $YB2full = cmp2 \cdot TC2 \qquad -(2)$

YBfull : This is asserted when both YB1 and YB2 memories are full and no more data can be written into YB memory. i.e.

$$YBfull = YB1full \cdot YB2full = cmpi \cdot TC1 \cdot cmp2 \cdot TC2 \qquad -(3)$$

YB1empty : This is asserted when YB1 memory is empty. It is ensured when terminal count TC1 is low and both read and write counters are equal. i.e.

$$YB1empty = cmpi \cdot \overline{TC1} \qquad -(4)$$

YB2empty : Similarly $YB2empty = cmp2 \cdot \overline{TC2} \qquad -(5)$

YBempty : This is asserted when both YB1 and YB2 memories are empty and no more data can be read from YB memory. i.e.

$$YBempty = YB1empty \cdot YB2empty = cmpi \cdot \overline{TC1} \cdot cmp2 \cdot \overline{TC2} \qquad -(6)$$

When YB memory is reset both read and write counters point to location zero. First write is always in YB1 and first read (before any data has been written in the memory) is blocked by skipping of the sequencer's cycles. Writing continues in YB1 until simultaneous read and write occur. In the later case , writing is started in YB2 and reading is done from YB1. When the YB1 is empty , read cycles are started from YB2 and writing in YB2 continues until simultaneous read/write occur.

Following are the conditions when writing switches from YB1 to YB2 (YB2 to YB1)

i) Occurrence of simultaneous read and write when both reading and writing was on in YB1(YB2).

OR

ii) When YB1(YB2) is full.

Switching from YB1 to YB2 (YB2 to YB1) in read cycles takes place when YB1 (YB2) is empty.

The logic developed is as follows.

$$wryb1 = wryb \cdot \overline{WRS1\_S2} \cdot (cmp1.TC1 + rdyb \cdot \overline{RDS1\_S2}) + wryb \cdot \overline{WRS1\_S2} \cdot (cmp2 \cdot TC2 + rdyb \cdot \overline{RDS1\_S2}) \quad -(7)$$

Here the term cmp1.TC1 indicates , whether the YB1 is full or not. 'rdyb' indicates simultaneous read. Thus first expression indicates that YB1 is written when the previous write was in YB1 and YB1 is not full and , there is no simultaneous read in the YB1 memory. The second expression indicates that YB1 is written when the previous write was in YB2 and YB2 is full or there is a simultaneous read of YB2 memory.

Similarly

$$wryb2 = wryb \cdot \overline{WRS1\_S2} \cdot (cmp2.TC2 + rdyb \cdot \overline{RDS1\_S2}) + wryb \cdot \overline{WRS1\_S2} \cdot (cmp1 \cdot TC1 + rdyb \cdot \overline{RDS1\_S2}) \quad -(8)$$

For reading of YB

$$rdyb1 = rdyb \cdot \overline{RDS1\_S2} \cdot \overline{YB1empty} + rdyb \cdot RDS1\_S2 \cdot YB2empty \quad -(9)$$

Similarly

$$rdyb2 = rdyb \cdot \overline{RDS1\_S2} \cdot \overline{YB2empty} + rdyb \cdot \overline{RDS1\_S2} \cdot YB1empty \quad -(10)$$

IFU's sequencer's cycle is skipped when the IFU tries to read empty Yb memory.

The EPROM controller implements equations 1 to 10.

d) DATA MEMORY :

For a large data input , on which computation is to be performed, X memory size may not be sufficient. In that case the data can be stored in the data memory (fig. 3.3) and when the X memory is empty , X memory can be loaded from the data memory. The address for the data memory is generated by the address generator in the IFU.

3.1.4 Run Time Flow Control :

Flow control for the inter cell communication channel is assumed to be implemented in the hardware. When a cell tries to read an empty queue, all the computations and sequencer's sequencing are blocked until a data item arrives in the queue. Similarly when a cell tries to write to a full queue of a neighbouring cell , the sender is blocked until a data item is removed from the full queue. The blocking of the cells is transparent to the program and hence the programmer need not keep track of data flow through the array on cycle by cycle basis, while programming. The state of all the computation units on the data path freeze for the duration the cell is blocked. All other cells in the SASP array continue to operate normally. The data queues of a blocked cell are still able to accept inputs.

To achieve this run time flow control, the following signals are generated.

a) IFU-cell communication control signals: [fig. 3.5]

Xqfull -> To indicate the IFU that no more data can be written into the X queue of cell i. If the IFU tries to write into X queue , it will be blocked.

Yqfulli -> To indicate Y queue of cell i is full.

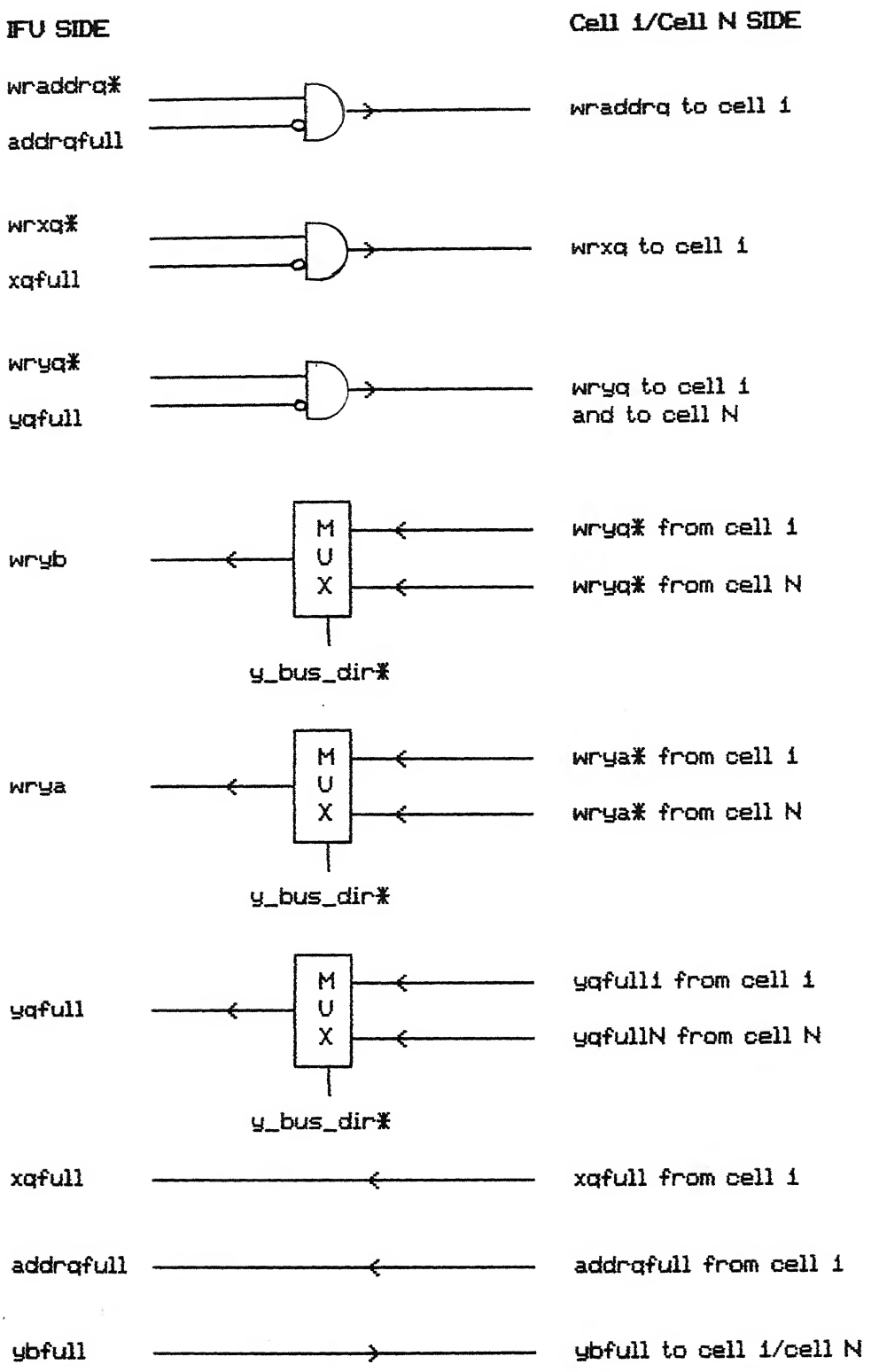


Fig. 3.5 IFU - CELL Communication Control signals.

YqfullN -> To indicate Y queue of cell N is full.

One of the above two Yqfull signals is selected by the multiplexer as per the Y bus direction.

Addrqfull -> To indicate address queue of cell i is full.

wrya -> Write YA memory signal from both cell i and cell N . One signal is selected by the mux.

wryq from cell -> Write Y queue signal from the celli/cellN which acts as write YB signal for YB memory in the IFU..

wrxq* -> Signal from the IFU microcode to cell i to write X memory data into X queue of cell i.

wryq* from IFU -> Signal from the IFU microcode to cell i/cell N to write into Y queue of cell i/cell N.

wraddrq* -> Signal from the IFU microcode to cell i to write into the address queue of cell i.

ybfull -> Signal to cell i/cell N to indicate YB memory is full.

b) CELL - CELL communication control signals : (For cell n)

The signals coming to the cell n are as follows.

xqfull -> To indicate that X queue of cell n+i is full.

yqfulln+i -> To indicate that Y queue of cell n+i is full.

yqfulln-i -> To indicate that Y queue of cell n-i is full.

One of the above two yqfull signals selected as per Y bus direction.

addrqfull -> To indicate that Address queue of cell n+i is full.

wryqn+i -> Write Y queue signal from cell n+i.

wryqn-i -> Write Y queue signal from cell n-i.

One of the above two signals acts as 'wryq' signal for the present cell's Y queue

as per the Y bus direction.

wrxq -> Signal from cell n-1 to write into X queue of present cell.

wraddrq -> Signal from cell n-1 to write into Address queue of present cell.

There are some 'local' control signals to check the data flow.

xqempty -> Indicating that X queue of the present cell is empty and a read X queue attempt will block the cell.

yqempty -> Indicating that Y queue of the present cell is empty.

addrqempty -> Indicating that Address queue of the present cell is empty.

Thus for a cell, cycle is skipped (cell is blocked) , when 'cycleskip' signal is HIGH , where,

cycleskip = xqempty . rdxq* + yqempty . rdyq* + addrqempty . rdaddrq*
+ xqfull . wrxq* + yqfull . wryq* + addrqfull . wraddrq*

3.2 The SASP Cell Unit:

Fig. 3.6 shows the block diagram of the SASP cell. Each cell is implemented as a programmable horizontal machine , with its own microsequencer and program memory. The microinstruction details for the cell unit are given in the Appendix C. The data cross bar of the cell provides very high intra cell bandwidth and X , Y channels with their associated queues provide high inter cell bandwidth.

3.2.1 Microengine and Address generation unit :

The structure of the microengine is similar to the microengine of the IFU. Here the FLAG input to the sequencer has the following sources.

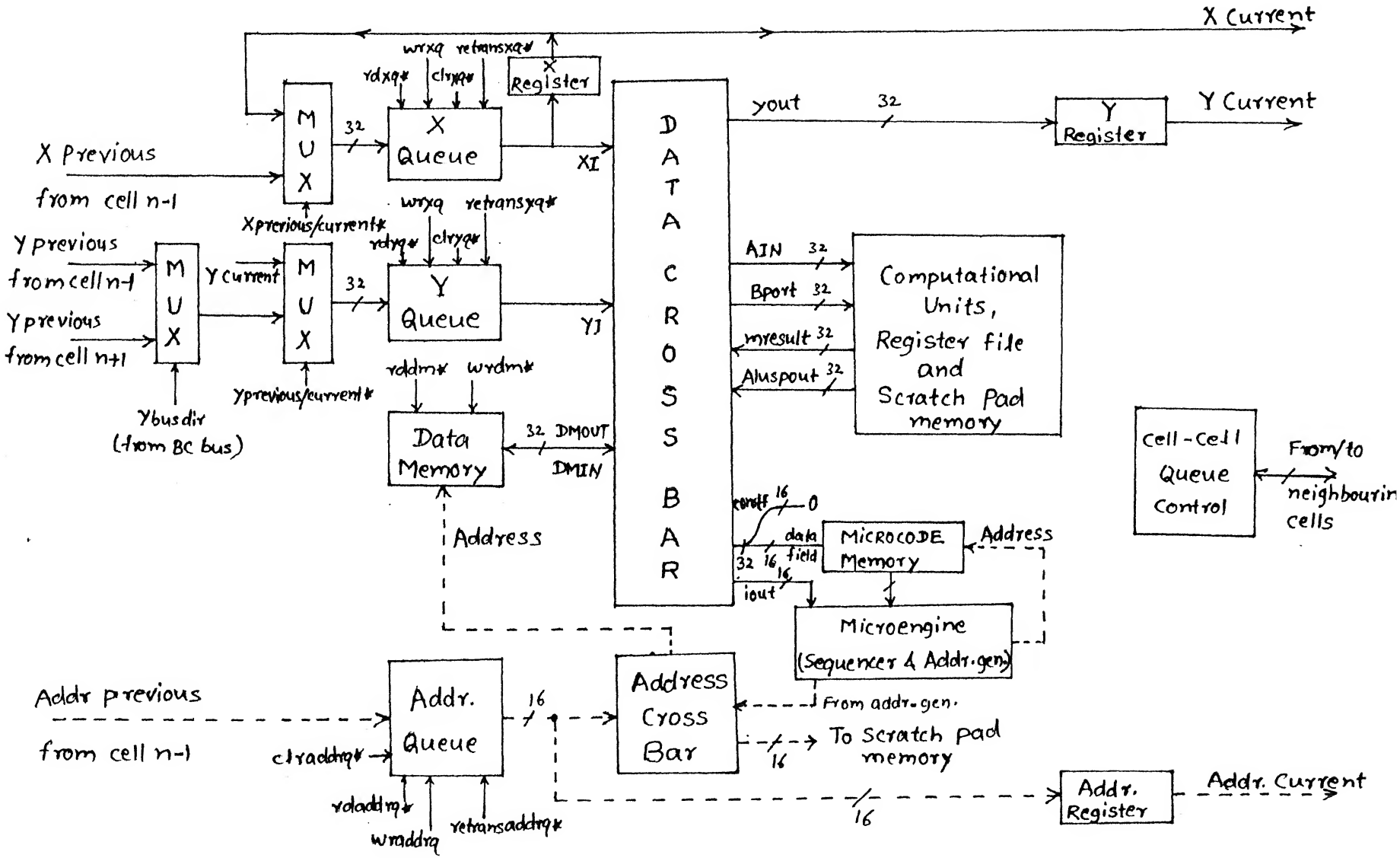


Fig. 3·6 SASP Cell Block Diagram.

cmpz , flg as described in the section 3.1.2 (fig 3.2).

OVRFLO , UNDFLO , INVALOP -> These are output signals from the ALU chip.

Lessthan , greaterthan, equal -> These signals are derived from the output signals of ALU chip for the compare instruction.

Data independent addresses are generated in the IFU, whereas data dependent addresses are generated in the SASP cells. Thus the address generator is used as the local address generation unit.

Addresses to local data memory and scratch pad memory are fed from the address cross bar , which has inputs from the address queue and address port of the address generator ADSP-1410. The sequencer ADSP-1401 and address generator ADSP-1410 data ports get the data from the data cross bar. Thus the data to the data ports can be from the constant (data) field of the microinstruction or from any of the inputs to the data cross bar. Since the data ports of sequencer and address generator are 16-bit wide and data cross bar output is 32-bit , only least significant 16-bits are considered. This feature enables the address generator unit to calculate data dependent addresses.

3.2.2 Inter cell Communication :

Each cell can communicate with its left and right neighbours through X, Y and Addr bus. A queue is associated with each channel to increase the inter cell bandwidth.

a) X queue -> The input to the X queue can be from the cell n-1 (x previous) or from the output of X queue itself (x current). This local feedback can be used to give delay to the X data , which is also useful to simulate multiple cells using

single cell. The microcode signal inputs to the X queue are , rdxq* - To read X queue , wrxq* - signal from cell n-1 to write the queue , clrx* - to clear read and write counters of the queue, retransx* - to retransmit the X queue data from the first physical location. (read counter is reset to zero.)

The details of the implementation of the queue structure is shown in fig. 3.7. It uses the FIFO CY7C420 . The queue consists of a Dual port RAM . The following operations can be performed using the various control signals.

Resetting the FIFO : A master reset $\overline{MR}$, causes the FIFO to enter the empty condition signified by the empty flag ($\overline{EF}$) being active. i.e. both read and write counters are reset to zero .

Writing data to the FIFO : The availability of an empty location is indicated by the inactive state of the Full flag ($\overline{FF}$). A falling edge of write ($\overline{W}$) initiates a write cycle.

Reading data from the FIFO : The falling edge of Read ($\overline{R}$) initiates a read cycle, if the empty flag ($\overline{EF}$) is not active. The falling edge of $\overline{R}$ during the last read cycle before the empty condition triggers the $\overline{EF}$ to active , provibiting any further read operation , until a valid write.

Retransmit : The retransmit feature is beneficial when transferring packets of data. A low pulse at $\overline{RT}$ resets the internal read pointer to the first physical location of the FIFO. The write pointer is unaffected.

After a retransmit cycle , previously read data may be reaccessed beginning with the first physical location.

The output signals from the FIFO CY7C420 $\overline{EF}$ and $\overline{FF}$ are used to signal the queue condition (empty or full). clrx* signal from the microcode memory gives the master reset ($\overline{MR}$) to the chip and retransx* signal acts as retransmit ($\overline{RT}$) signal to the chip.

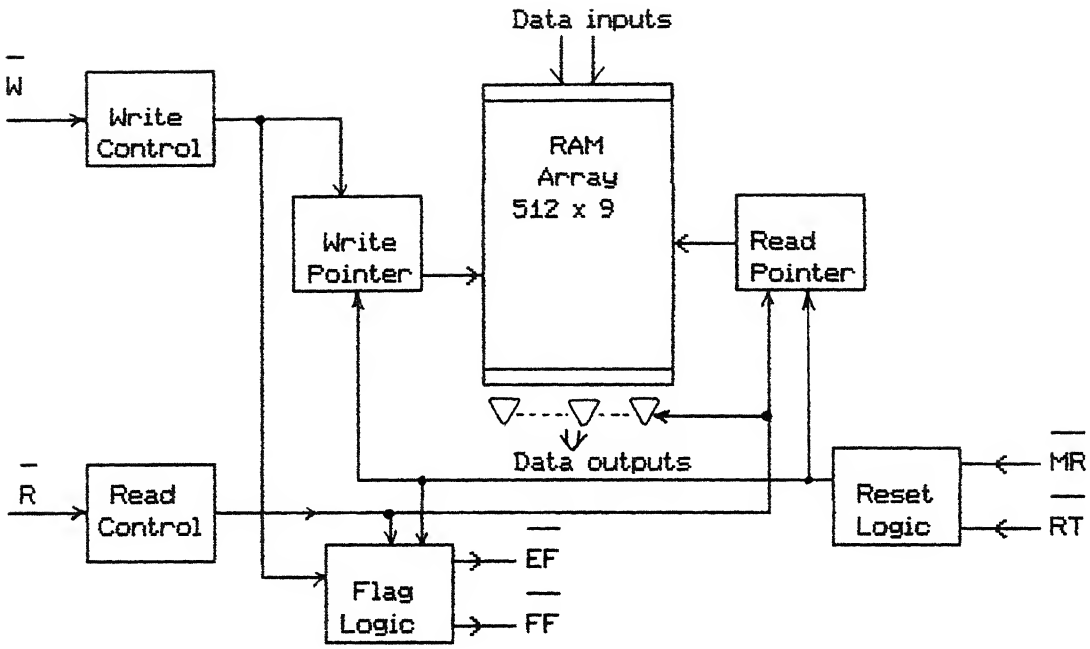


Fig. 3.7 Logic Diagram of CY7C420 FIFO

b) Y queue -> The input to the Y queue can be from the cell n-1/cell n+1 (depending on the Y bus direction) i.e. Y previous or from the local feedback. The signals to the Y queue are similar to X queue and it is also implemented using CY7C420 FIFO chip.

c) Addr queue -> The input to the Address queue is from the previous cell (cell n-1) and local feedback is not provided. The structure is same as X queue and Y queue.

The X register in the X data path (fig. 3.6) is strobed with the X data output of X queue by the signal rdxq* . And when wrxq* signal is given by the cell n and if X queue of cell n+1 is not full , this strobed data is written into the X queue of cell n+1.

Similarly the Y register is strobed with the data from the yout cross bar output. It is strobed only when the yout port of the data cross bar is enabled. The data in the Y register is written into the Y queue of cell n-1/cell n+1 , in a way similar to X queue.

The addr register function is same as X register.

These registers are used because , considering the distance between two cell unit boards, the reading of the local queue and writing of the queue of neighbouring cell in a single cycle is difficult to realize in the hardware.

3.2.3 Cross bar :

Internal data bandwidth is often the bottleneck of a systolic cell. In the cell the two floating point chips can consume up to four data items and generate two

results per cycle. Cross bar connecting various data storage blocks support this high data processing rate. Moreover, the use of cross bar leads to complication when compared to bus based systems. There are 5 input ports, 4 output ports and one bidirectional port. An output port can output data from any of the inputs , irrespective of other outputs.

The input ports are

XI -> from X queue.

YI -> from Y queue.

constf -> from the constant (data) field of the microcode.

mresult -> from the multiplier output.

alu_spout -> from the ALU output.

The bidirectional port is to/from the data memory.

The output ports are

Ain -> to ALU input.

Bport -> to B port of the register file.

iout -> to the internal data bus of the microengine.(16-bit bus)

yout -> input to the Y register.

The cross bar can be realized using PALS or using multiplexers. The input port, for an output port is selected using a 3-bit control signals from the microcode.

For example, for the output port yout , to select various inputs, the signals which are encoded in 3 microcode bits, are as follows.

xbyout_xi -> input from XI i.e. X queue.

xbyout_yi -> input from Y queue.

xbyout_dmout -> input from data memory.

xbyout_aluspout -> input from ALU output bus.

xbyout_mresult -> input from multiplier output.

xbyout_constf -> input from the constant (data) field of the microcode.

xbyouttri -> Tristate the output.

3.2.4 DATA STORAGE UNITS :

The local data storage units includes a data memory , a register file and a scratchpad memory.

a) Data Memory : The local data memory can be accessed in every clock cycle. It is generally used for loading weights, coefficients , which are not transmitted during the execution of an algorithm. Intermediate results can also be stored.

b) Register file : [ADSP-3128 (128 * 16)] (fig. 3.8)

It is a versatile data storage component which greatly expands the computational bandwidth of a fast arithmetic processor. The ADSP- 3128 also simplifies processor design by permitting flexible data routing through its five 16-bit ports : two input ports , two output ports and a bidirectional port. Two register files are used "horizontally" yielding 128 words of 32-bit storage. The five ports allow six 32-bit transfer operations per cycle for single precision mode using two chips.

Register to register transfers are made via the bidirectional E data port. Writes to the RAM occur in clock HI. Note that data written in clock HI is available to be read in the same clock cycle.

The arrangement of the register file ports is as shown in the fig 3.8. Ports D and C provides data to the ALU and multiplier chips respectively. Port E is

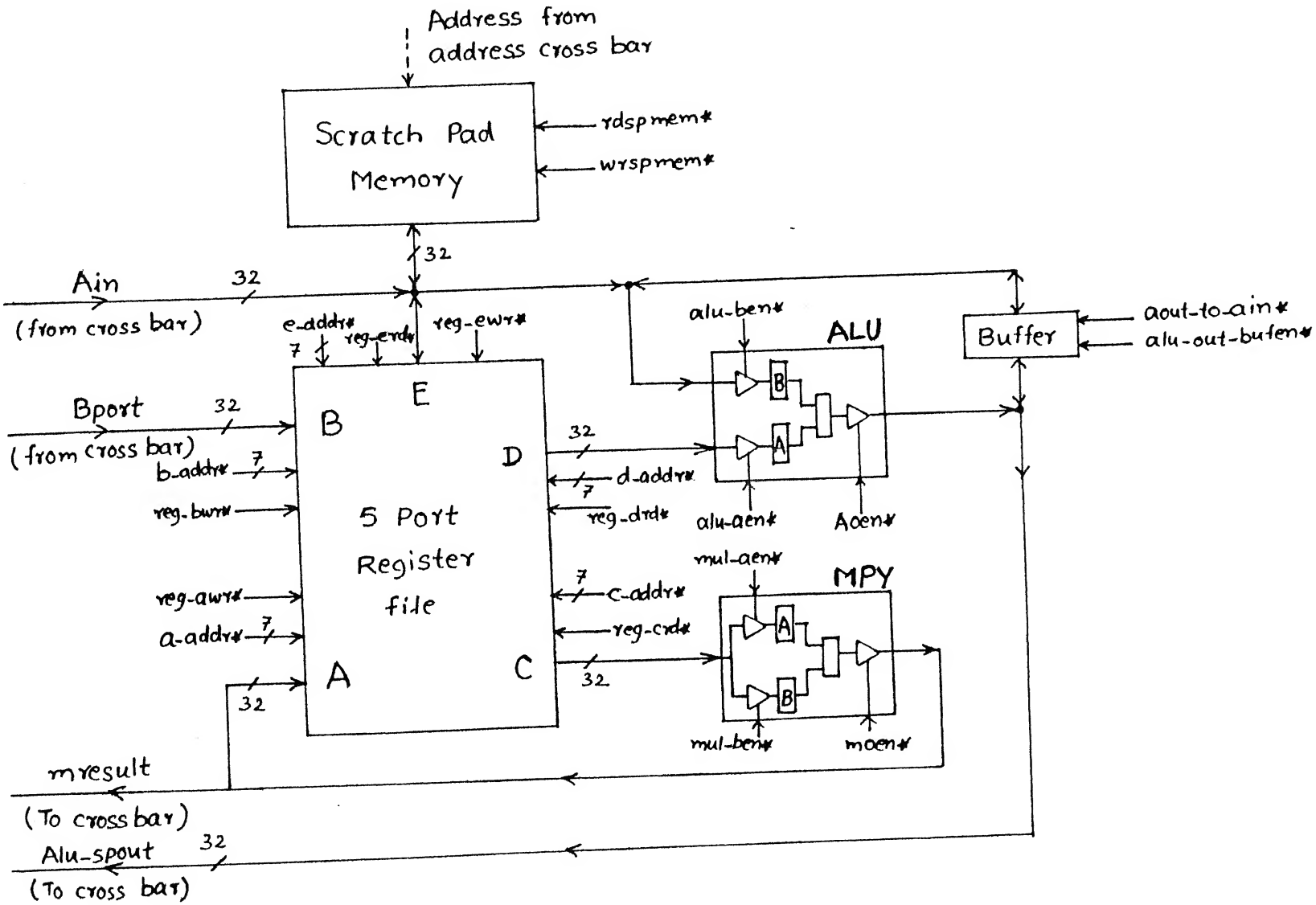


Fig. 3.8 Computational units and the register file of the SASP cell.

bidirectional and can be used to receive input from the cross bar , scratch pad or from the ALU output. It can act as output for writing into the scratchpad , into the ALU port , or input to crossbar through the bidirectional buffer. The input port A accepts the multiplier output.

Thus the register file can be used for storing the intermediate results and to route the data/results to/from the various units.

c) **Scratchpad Memory** : The scratchpad memory can be used to hold scalars , floating point constants and small arrays. The addition of the scratchpad increases memory bandwidth and improves through-put for those programs operating mainly on local data.

Addresses to the data and scratchpad memory come from the address crossbar , whereas those for register file come from control bits of the microcode.

3.2.5 Computational Units : [Fig. 3.8]

The cell is using a floating point multiplier ADSP-3210 and a floating point ALU ADSP-3220 confirming to IEEE standard 754. 32-bit twos-complement fixed-point operations are also supported by these chips. Both these units have two - stage pipelining.

The ADSP-3210/3220 share a common architecture [Fig. 3.9]. The input registers can be read to the chip's computational circuitry as they are loaded. At the end of first processing clock cycle, partial results are clocked into a set of internal pipeline registers. At the end of second processing cycle, results are clocked into an output register. The contents of the output register can then be driven off chip.

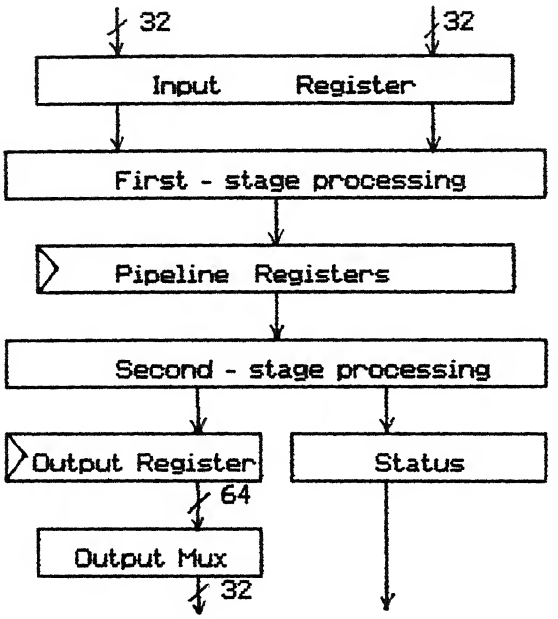


Fig. 3.9 ADSP 3210/3220 Generic Architecture

Because all input and output data are internally registered and because of the single level of internal pipeline registers, operations can be overlapped for high levels of pipelined throughput. The table illustrates a typical sequence of pipelined throughput.

Time (cycles)	Load input data	first stage	Second stage	Output result
1	Data set A			
2	Data set B	Data set A		
3	Data set C	Data set B	Data set A	
4	Data set D	Data set C	Data set B	Data set A

Single - precision floating point data format is as follows.

sign	exponent (e)	fraction (f)
s	e7 e0	. f22 f0

The following mnemonics indicate types of floating point numbers in the computations.

Mnemonic	Exponent	Fraction	Value	Name
NAN	255	non-zero	undefined	not - a -number
INF	255	zero	$(-1)^S$(infinity)	infinity
NORM	1 - 254	any	$(-1)^S(1.f)2^{e-127}$	normal
DNRM	0	non-zero	$(-1)^S(0.f)2^{-126}$	denormal
ZERO	0	zero	0	zero

STATUS FLAGS : These chips generate on dedicated pins the following exception flags specified in the IEEE standard : Overflow (OVRFLO), Underflow (UNDFLO), and Invalid operation (INVALOP).

Conditions that cause the assertion of INVLOP are :

- NAN input to computational circuitry.
- Multiplication of either +/- INF by either +/- ZERO.

For comparison operations in the ALU , the OVRFLO , UNDFLO, and INVALOP status outputs are used to indicate four comparison conditions.

-"Less than" is signaled by the assertion of UNDFLO (while OVRFLO is low)

-"Greater than" is signaled by the assertion of OVRFLO (while UNDFLO is low)

-"Equal" is signaled by not asserting either UNDFLO or OVRFLO.

-"Unordered" is signaled by the assertion of INVLOP caused by attempting a comparison with at least one NAN operand.

Instructions And Operations : The ADSP-3210 multiplier executes the same instruction every cycle : multiply. It need not be specified explicitly in the microcode.

The ADSP-3220 ALU , in contrast to the multiplier is instruction driven with the operation specified by I8-0.

Since only two registers A0 and B0 of both the chips are used , only two signals are required (Aen* and Ben*) to load the input registers. This is done to simplify the microcode. oen* signal is used to enable the output port. For the multiplier, the microcode signal msp*/mfixed* specifies either a single precision floating point multiplication or a fixed point multiplication of the two register inputs A and B. For the ALU the instruction field is used to specify these options.

The detailed description of multiplier and ALU (ADSP-3210/3220) chips is given in the Appendix D.

4. SIMULATOR

4.1 INTRODUCTION :

Simulation is the process by which understanding of the behaviour of an already existent or planned physical system is obtained by observing the behaviour of the model representing the system. A simulation study must have a purpose and there are many reasons why simulation is valuable. For example , simulation may be performed to check and optimize the design of a system before its construction , thus helping to avoid costly design errors and ensuring safe designs. Other purpose include analysis, performance evaluation, cost effectiveness, forecasting , safety, teaching , decision making.

Simulation is thus a very widely used technique. However , simulation of a complex processor or a complex computer system as a tool for software development is of recent origin. The main uses of such a simulator are ease of software development , debugging, testing , modifications and evaluating the performance of the system. The main aim of the simulator described in this chapter is to provide a facility to develop signal processing algorithms on the SASP system and evaluation of the performance of the SASP system. The user has almost all the freedom for software development as if he is working with a hardware system. The simulator program uses the object code assembled by the meta-assembler. It is not a real time simulator. Simulation is cycle by cycle and one cycle is equivalent to one clock cycle in the hardware system. In the hardware one cycle is equal to 10 microseconds for a 10 Mhz clock.

4.2 SIMULATOR BLOCKS:

The main building blocks of the simulator are the simulation of the various components of SASP. The following chips have been simulated.

i) Sequencer ADSP-1401

ii) Address generator ADSP-1410

iii) ALU ADSP-3220

iv) Multiplier ADSP-3210.

In addition a FIFO queue has also been simulated as it is extensively used in the SASP architecture. Each of these blocks is explained below.

4.2.1 Simulation of the Sequencer [ADSP-1401]:

The details of the chip are given in the Appendix D. All the features of the chip are simulated. Each of the sequencer instructions is simulated using a separate routine. And the registers and signals of the chip are expressed in the form of global variables. The simulation process is as follows.

In an instruction cycle, first the subroutine for the instruction corresponding to the opcode presented to the sequencer is executed. Then the program checks for an intrrupt request. If a request is pending the program counter is loaded with the interrupt vector. At the end , the program checks the interrupt pins and internal interrupt sources for an interrupt. If any of these sources is active a request is raised. A flowchart is given in figure 4.1.

The ADSP-1401 processes eight external and two internal interrupts. The two internal interrupts are reserved for stack overflow - IR9 and counter underflow - IR0 (See Appendix D). The simulator has internal cycle counters, which can be set to model interrupting devices. User can activate any one of the 8

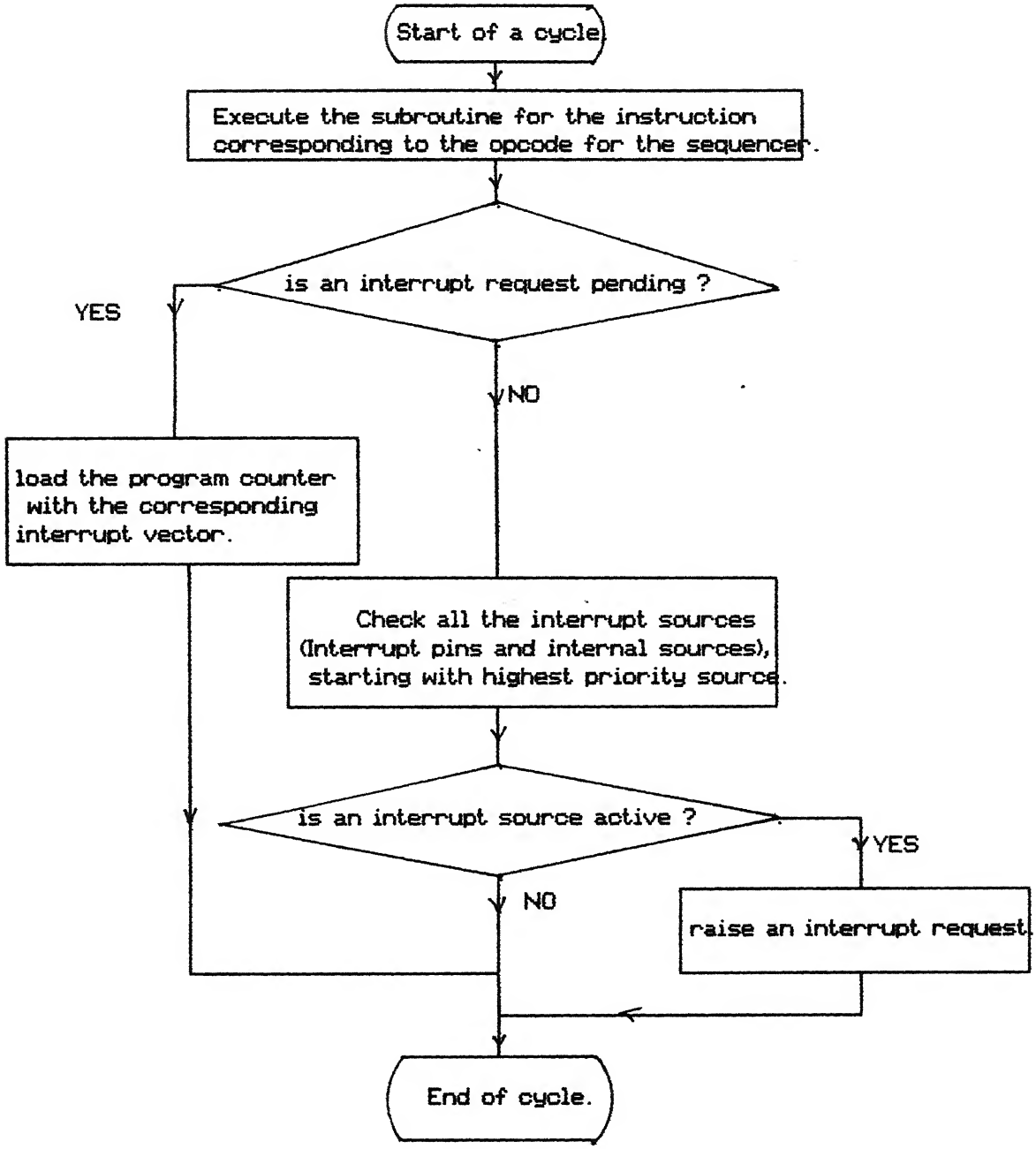


Fig. 4.1 Simulation of the Sequencer ADSP-1401.

external interrupting sources (IR1-IR8) by specifying time interval in cycles between two consecutive interrupts. When the interval time expires , an interrupt is issued at the corresponding level. To disable the interrupt , the interrupt period should be set to zero.

The other major features of the sequencer simulation are,

1) Internal 64-word RAM implementing two distinct stacks: a subroutine stack and a register stack. When stack overflow is detected , interrupt IR9 is raised.

2) Four independent 16-bit counters are used for maintaining loops and event tracking.

3) When the sign bit of status register is set IR0 interrupt is raised.

4.2.2 Simulation of Address generator [ADSP-1410]:

The simulation process is similar to the simulation of ADSP-1401 sequencer. Here also in every cycle the subroutine for the instruction corresponding to the opcode for the address generator (in the microcode) is executed. The instruction set of the address generator is given in appendix C.

4.2.3 Simulation of ALU and multiplier chips(ADSP-3220 and ADSP-3210):

The two stage pipelining is simulated in the following way. There are two registers A and B for both ALU and multiplier. Let us define an array 'result' of dimension 2. Now the following table illustrates the typical sequence of pipelined operations.

Time (cycles)	Load input data in A and B Reg.	first stage result0= operation A and B	Second stage result1=result0	Output result= result1
1	Data set P			
2	Data set Q	Data set P		
3	Data set R	Data set Q	Data set P	
4	Data set S	Data set R	Data set Q	Data set P

Here also, for each instruction of ALU chip, one subroutine is executed , in the simulator.

4.2.4 Simulation of a queue structure (FIFO CY 7C420):

The logic of the simulation of a queue structure is similar to the logic used in the hardware implementation of YB memory, described in section 3.1.3.

The size of the queue is fixed by the value given in the architecture description file. The status of the queue is given by two flags, Qfull, Qempty. Again the variable 'TC' is used, which is set when the terminal count for write counter is reached and is reset to zero when the terminal count for read counter is reached. The flowcharts for the 'Read Process' and the 'Write Process' are given in figure 4.2 and figure 4.3 respectively..

4.3 SIMULATION PROCEDURE:

The simulator simulates the SASP system. An architecture description file is input to the simulator to start a simulator session. By reading the architecture description file , it configures itself to match the target system hardware. In the architecture file the user can specify the size of X queue , Y queue , Addr queue

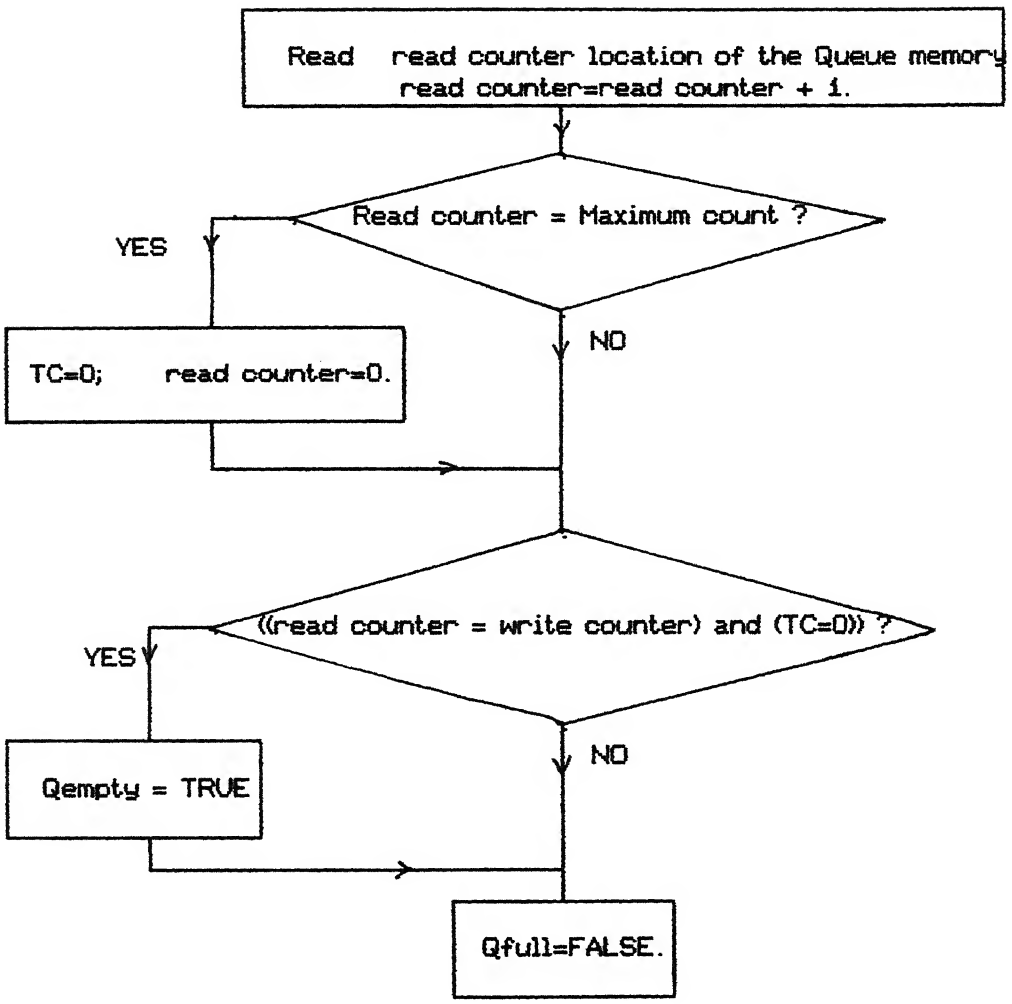


Fig. 4.2 Flowchart for reading a Queue.

data memory , program memory for a cell and X, YA , YB memory sizes for the IFU. Apart from these, the user can specify the array size , i.e. number of cells in the array.

The simulator program first reads an architecture description file and compact definition files for the microcodes of the IFU and a cell. After configuring itself, it prompts the user for a command. The flow of the simulator main program is as shown in figure 4.4.

The simulator contains a 'Command Table'. A binary search of the 'command table' is made to locate the command. If a valid command is found, then the corresponding subroutine is executed to serve that command.

Modularity of the program enables division of the problem into smaller tasks. After completing the desired task, the simulator returns to the simulator prompt mode until a logical termination of the simulator is asked by the user by giving the 'EXIT' command. The details of the commands are given in the appendix B.

The steps followed by the 'run' command subroutine are given in figure 4.5. In an instruction cycle, first some of the instructions from the microinstruction of IFU are executed. Then microinstructions for the array are executed starting from cell N to cell 1. At the end of the cycle remaining instructions for the IFU are executed. This sequence is followed to simulate parallel execution of all microinstructions for cells and the IFU.

For the single step routine , the first 5 (a to e) steps are followed once and then the simulator returns to the prompt.

The substeps followed by the first 3 steps are given in figures 4.6 , 4.7 and 4.8.

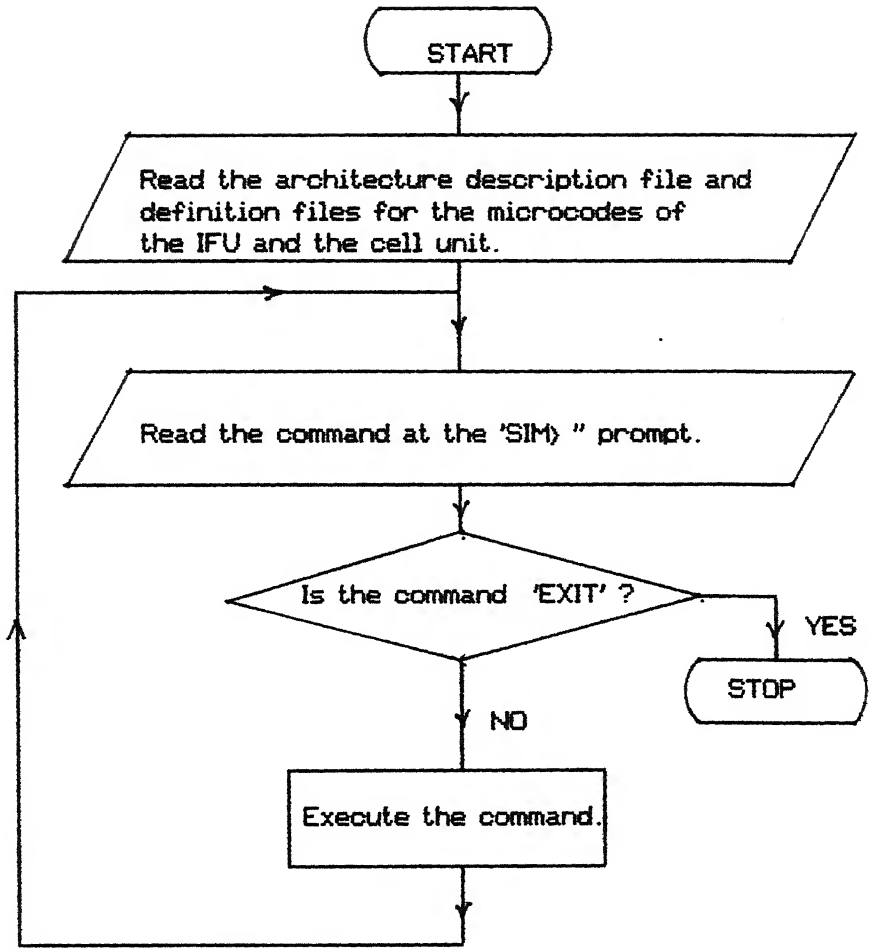


Fig. 4.4 The flowchart for the simulator program.

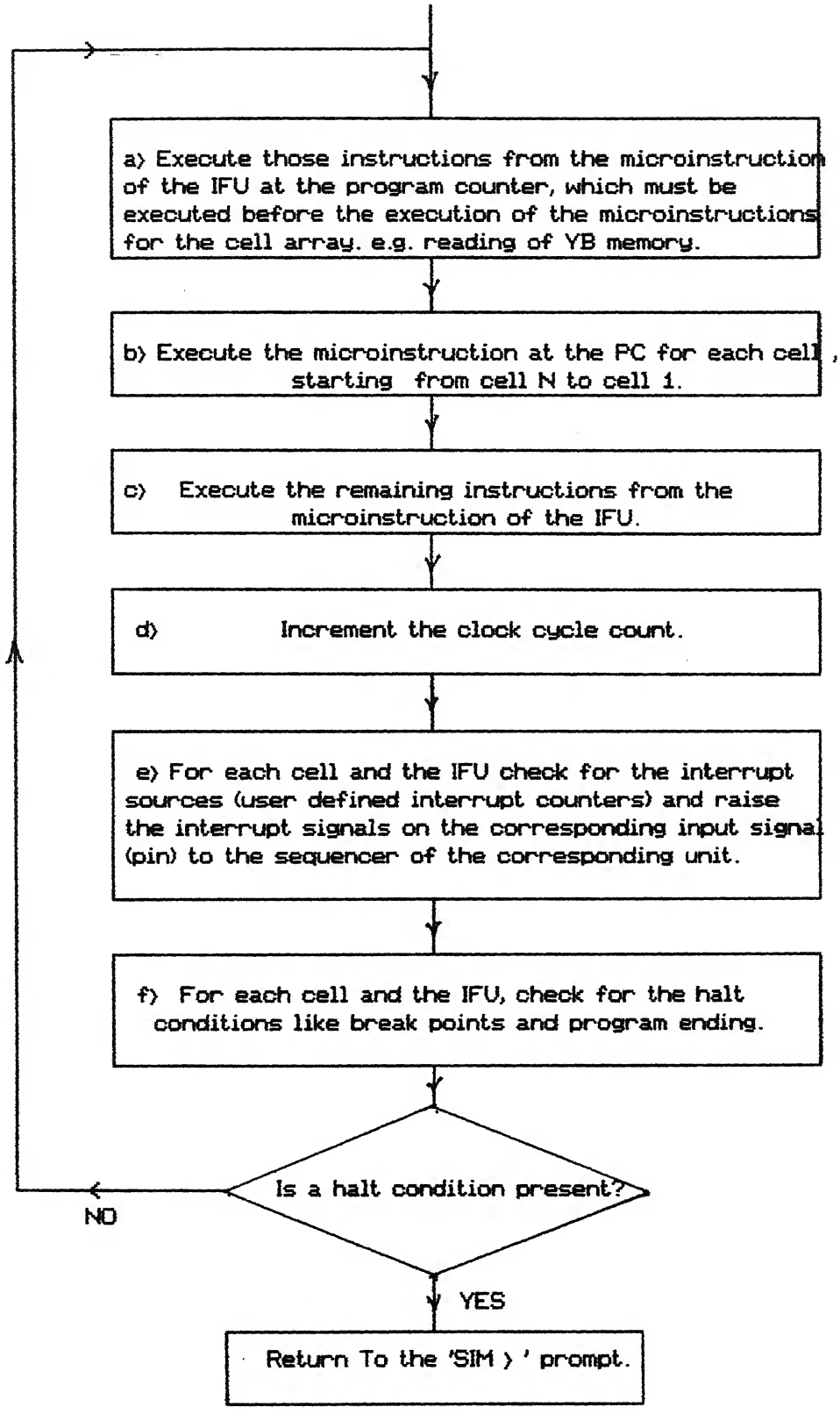


Fig. 4.5 Steps followed by the 'run' command subroutine.

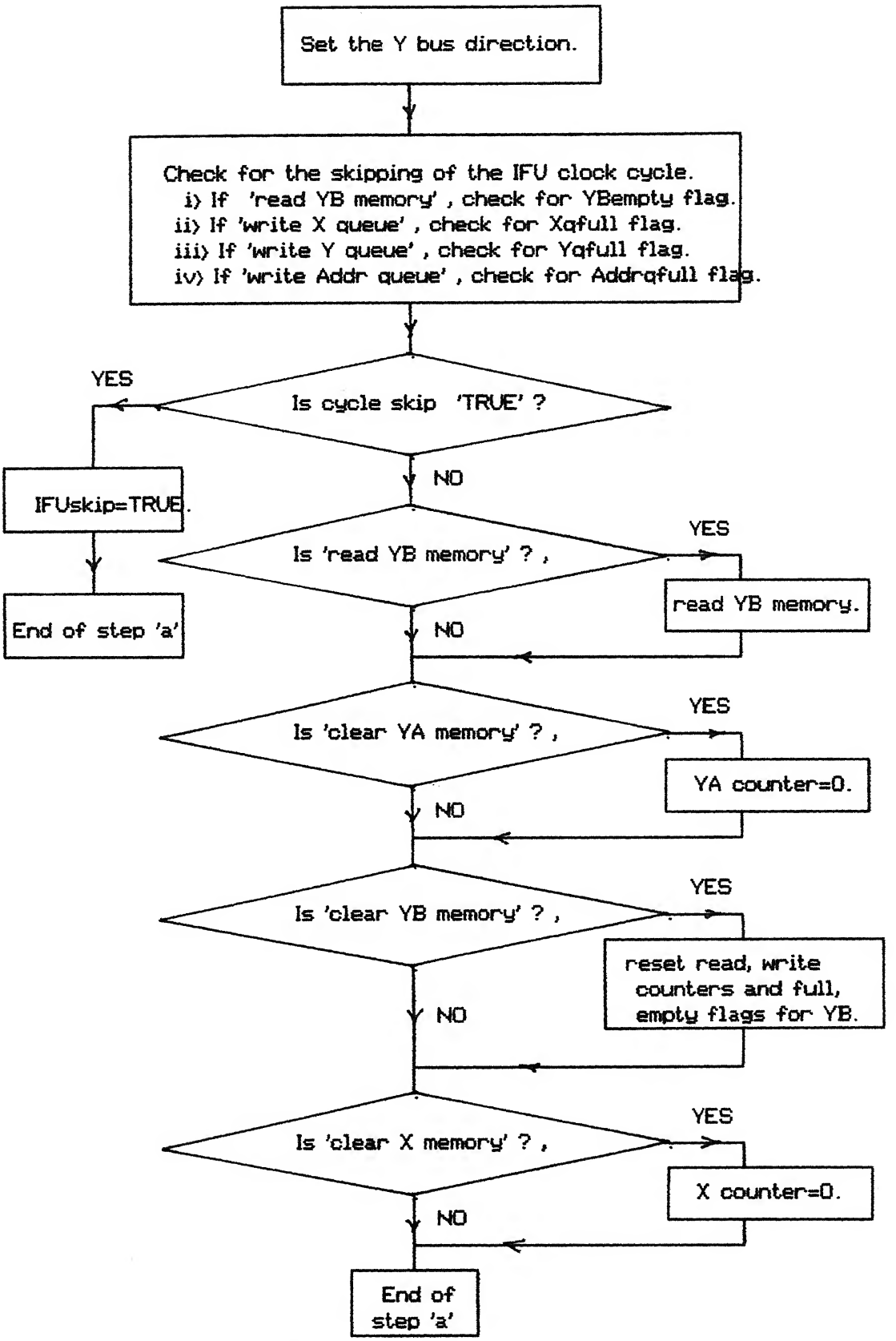


Fig. 4.6 Flowchart for step 'a' in fig. 4.5.

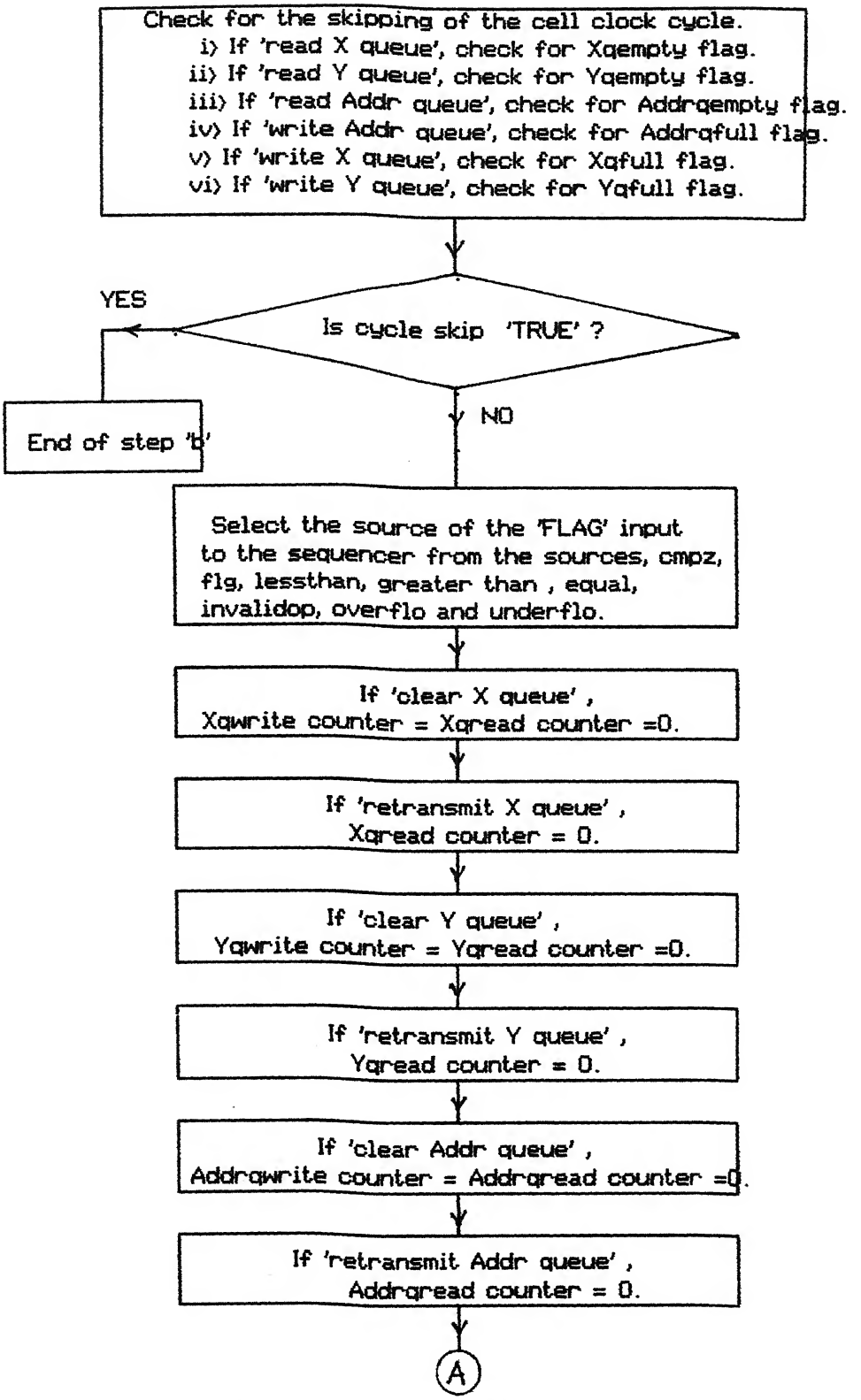


Fig. 4.7 Continued.

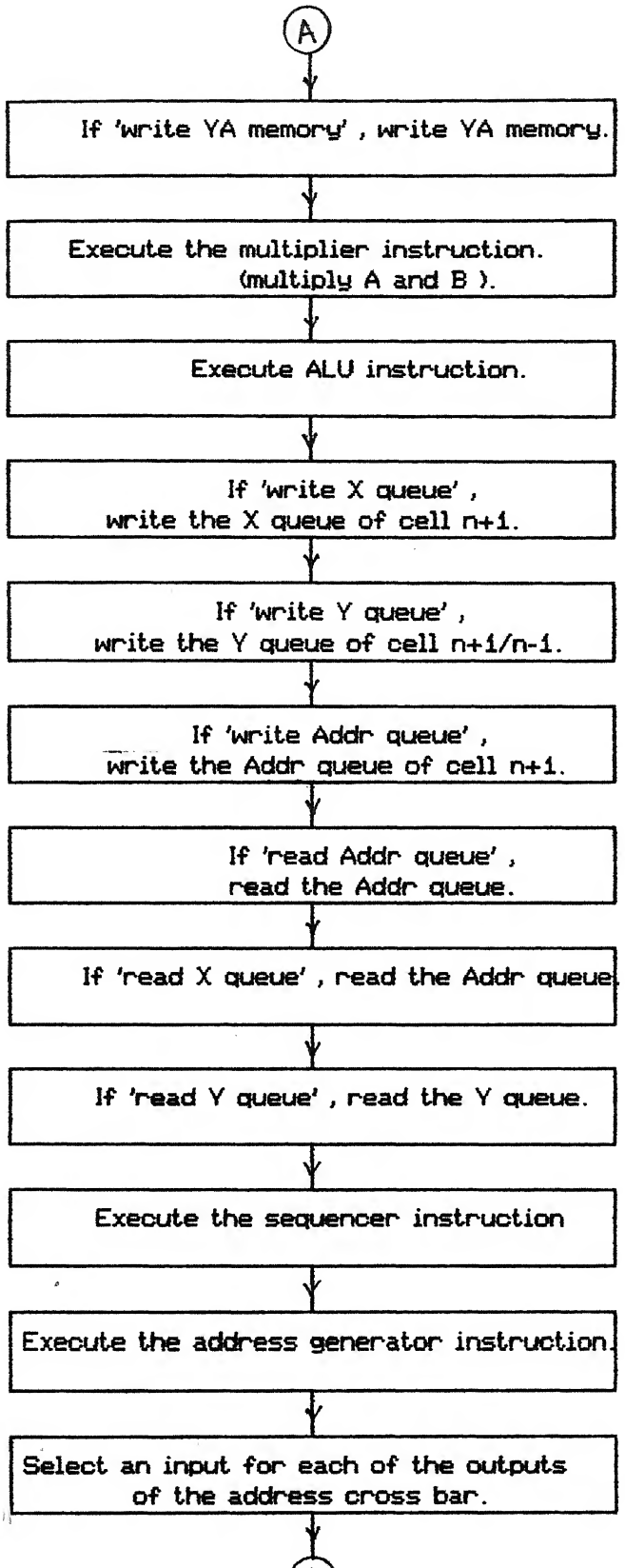


Fig. 4.7 Continued.

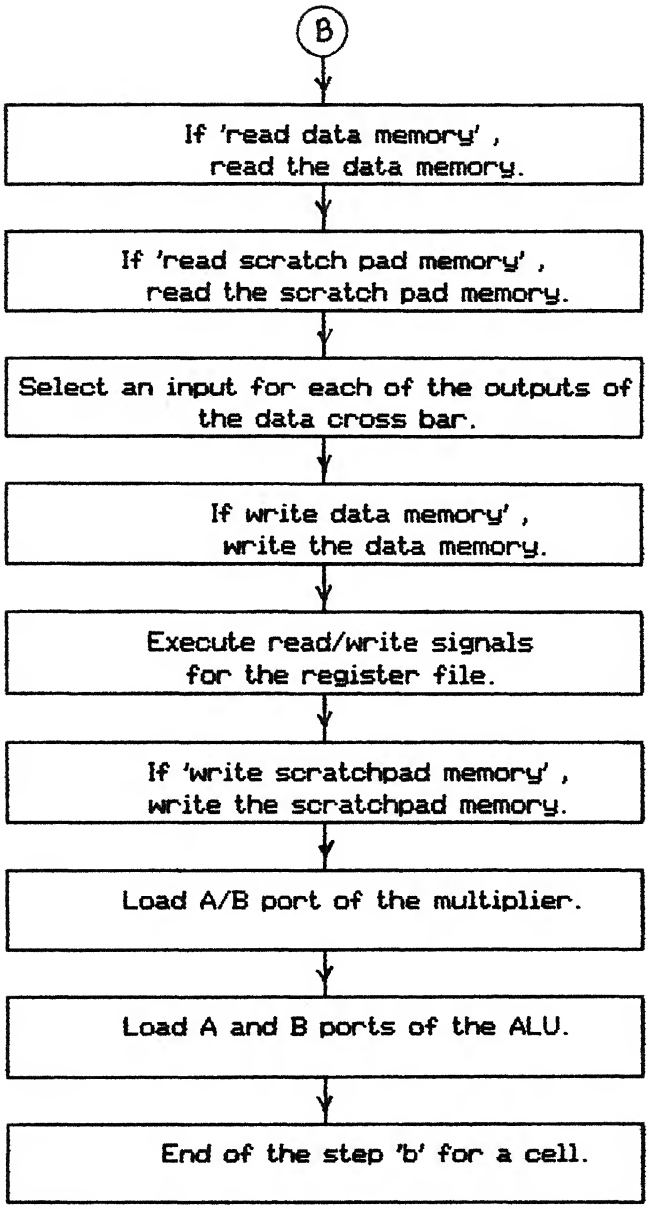


Fig. 4.7 Flowchart for step 'b' in fig. 4.5.

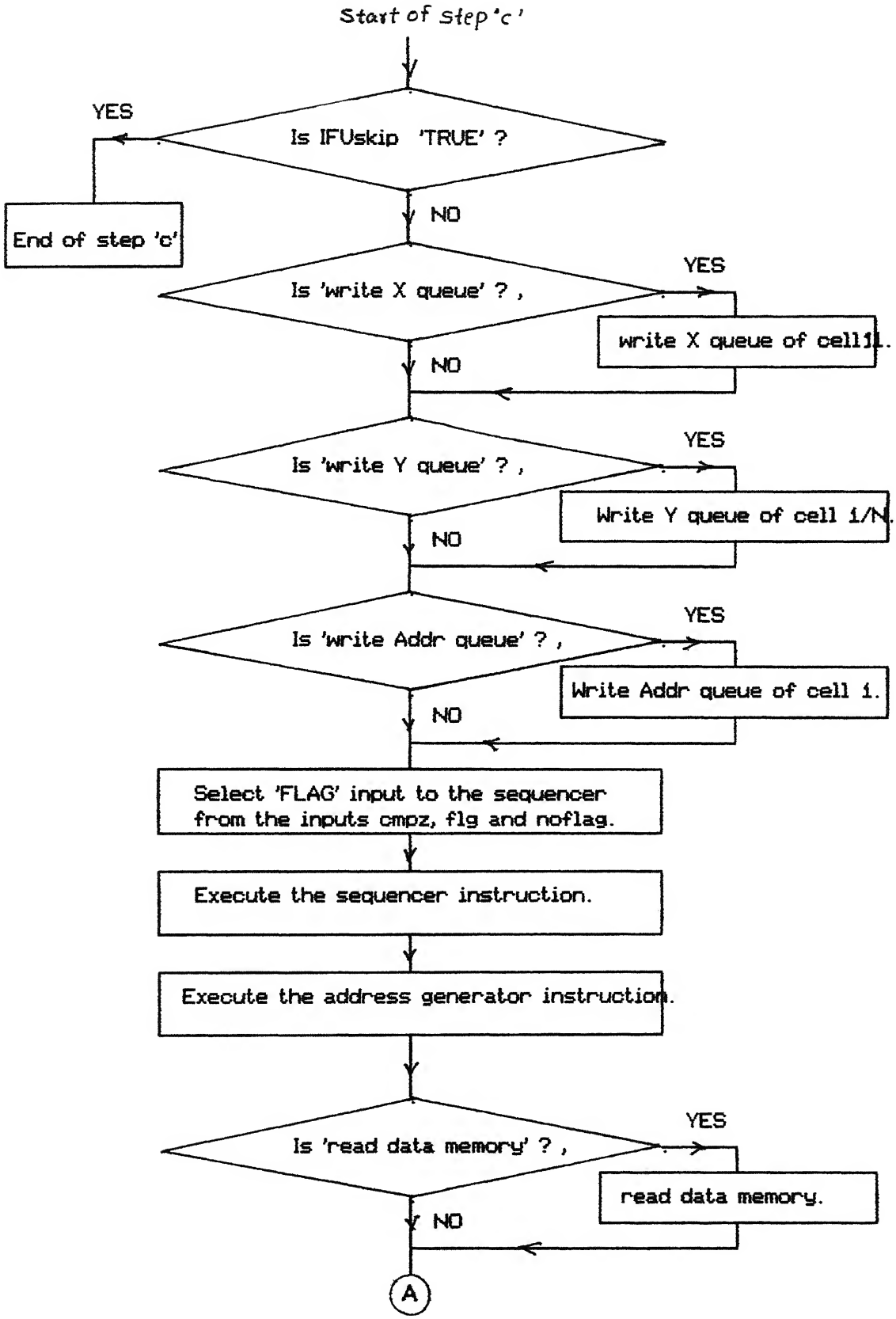


Fig. 4.8 Continued.

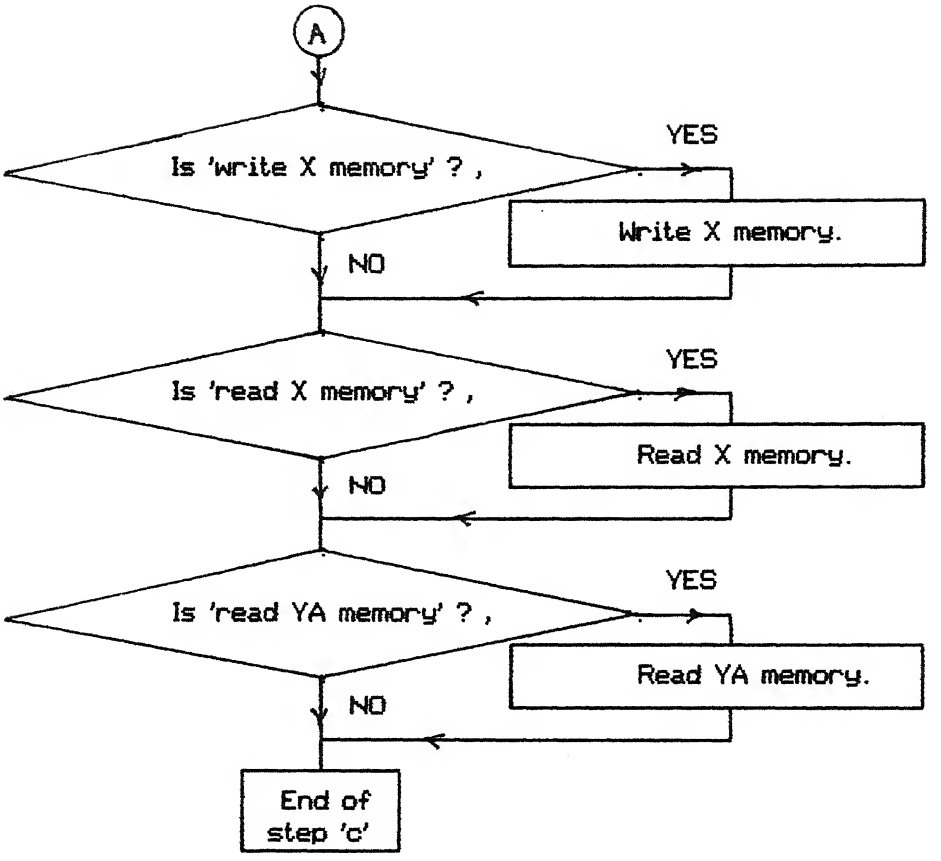


Fig. 4.8 Flowchart for step 'c' in fig. 4.5.

4.4 SPECIAL FEATURES:

1) During program development , the concept of modular programming has been strictly followed. Each instruction of a device (e.g. sequencer) is simulated by a separate subroutine.

2) Adequate documentation is provided in the source program to explain the working of the simulator in general and constituent subroutines in particular.

3) Addition of new instruction for a device is not restricted by the simulator. For addition of a new instruction , its name and opcode should be added in the structure for the list of instructions and a subroutine is to be added to execute the instruction.

4) Since each of the simulator block is simulated using separate subroutines and structures of variables , one can simulate some other architecture using these basic blocks, without any modifications in the software.

5) Total number of clock cycles required for the execution of a program is calculated to give the user an idea of actual execution time. This feature is useful for evaluating the performance of the system.

6) Error messages are flashed on the terminal.

7) Usual debugging facility is provided , for the development of the user programs. The facilities are

i) Load and display facilities.

ii) Trace facility.

iii) Break point facility.

iv) Execution facility, etc.

These facilities are described in the Simulator manual given in the Appendix B.

4.5 MODEL SESSION :

A model session for the matrix multiplication program is given. The algorithm of the program is described in the chapter 5.

```
Script started on Thu Jan 25 17:49:48 1990
$ ssim
_Arch_file: syst.arh

SIM>
SIM> lcode smul0.asm
ERROR :- Object file is not a valid file

SIM> lcode smul0.obj

SIM> lx -f
_Address: 0
_Length: 6
1 2 3 4 5 6

SIM>
SIM> element
Current Command cell: 0
_Number: 1

SIM> lcode
Obj file: smul1.obj

SIM> ldm -f
_Address: 0
_Length: 2
2 2

SIM>
SIM> element
Current Command cell: 1
_Number: 2

SIM> lcode
Obj file: smul2.obj

SIM> ldm -f
_Address: 0
_Length: 2
2
2
```

```
SIM>
SIM> element
Current Command cell: 2
_Number: 3

SIM> lcode
Obj file: smul3.obj

SIM> ldm -f
_Address: 0
_Length:
2 2

SIM>
SIM> run
Program terminated.

SIM> ybi -f
rdcntri=0
wrcntri=9
_Address: 0
0000h :6.000000e+00
0001h :1.400000e+01
0002h :2.200000e+01
0003h :6.000000e+00
0004h :1.400000e+01
0005h :2.200000e+01
0006h :6.000000e+00
0007h :1.400000e+01
0008h :2.200000e+01
0009h :0.000000e+00

SIM> exit
_Verify: y
$
script done on Thu Jan 25 17:53:21 1990
```

5. TESTING OF THE SIMULATOR

5.1 INTRODUCTION:

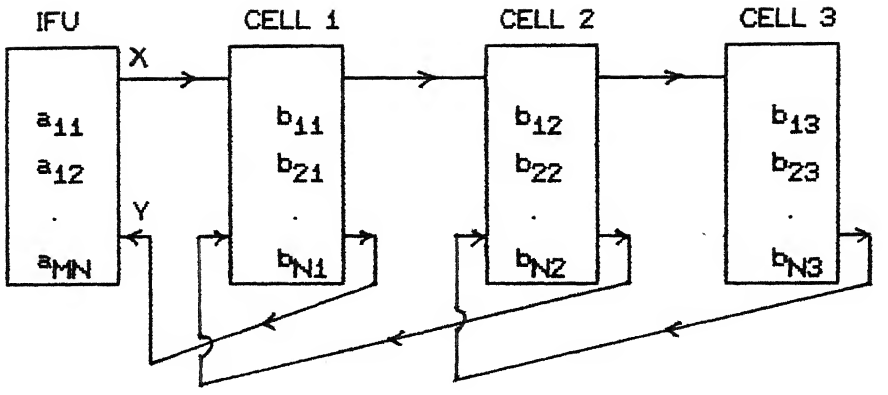
To test the simulator , two algorithms , for matrix multiplication and convolution are developed. The number of cells in the array has been fixed in the examples (equal to 3) . However this number is variable, and can be changed by changing value in the architecture description file. Both the algorithms are explained and the microprograms are also given. The microprograms are assembled by the meta-assembler described in chapter 6 . The simulator loads the object files and the data , before the start of the program execution.

5.2 MATRIX MULTIPLICATION ON THE SASP : [Fig. 5.1]

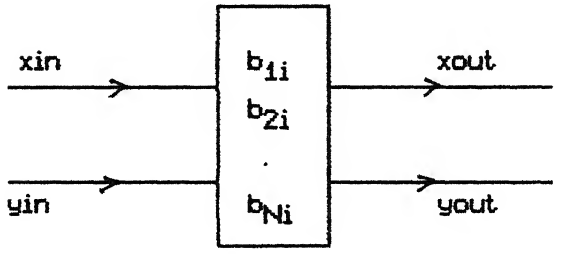
Let us consider two matrices A and B of size M x N and N x K respectively. Then their product C is a matrix of size M x K .

Matrix A is stored in the X memory of IFU row by row. Matrix B is stored in the data memory of the cells , one column in each cell. Therefore an array of K cells is required for this operation. The data is stored at the start in each cell and the IFU as shown in figure 5.1.

The IFU sends the X memory data on to the X channel sequentially. A cell n calculates a column of the result matrix using this X data on the X channel and sends the results to the cell n-1 on to Y channel using the reverse direction of the Y bus . At the end of the calculations, the cell sends the results written into its queue by the cell n+1 , to the cell n-1. The first cell in the array , effectively sends the results to the YB memory of the IFU.



CELL i SPECIFICATIONS



j=1 ; m=1

xout = xin = x_m

$Y_j = Y_j + x_m * b_{mi}$

m = m+1 , $1 \leq m \leq N$

m=1 ; yout= Yj ; j = j+1 $1 \leq j \leq M$

k=1

yout = yin

k= k+1 , $1 \leq k \leq (K-i)*M$

Fig. 5.1 MATRIX MULTIPLICATION ALGORITHM.

Thus the first element of the matrix C calculated by cell 1, is

$$c_{11} = a_{11}*b_{11} + a_{12}*b_{21} + \ldots\ldots\ldots + a_{1N}*b_{N1}$$

The reverse direction of Y bus is used, because the computations by cell 1 are over before the cell 3, and to make use of further cycles of the cell 1 for transferring of results to the YB memory, this feature is necessary .

The microprograms for the IFU and the cell 1 are given below. For other cells the program is same except, 1) the constant, 'cell_no' and 2) the last micro instruction at address 'end 1'. The constant, 'cell_no' is the serial number of a cell in the array. The last microinstruction for all other cells is 'jpcnf' instead of 'cont' for the cell 1. This instruction keeps the cell waiting, till the signal 'program terminated' comes from the cell 1. The model session for this program is shown in section 4.5.

```
;Program of Matrix multiplication for the IFU.
      M    equ 20
      N    equ 20

      wrcntr(C0) & 2  8000h+M*N/2-2 & ken & clrx & celln_1 & rdx
loop:  dccntr(C0) & rdx & wrxq & celln_1
      jda(sign) loop & ken & wrxq & rdx & celln_1
      jpcnf & celln_1
```

```
;THIS PROGRAM IS FOR MATRIX MULTIPLICATION ON THE SASP.
;IT IS TO  BE LOADED ON  EACH CELL.  NO. OF CELLS EQUAL TO K.
; A_matrix M*N
; B_matrix N*K
   N    equ   20
   M    equ   20
   K    equ    3
   cell_no  equ  1

      wrcntr(C1) & 2 8000h+M-2 & ken & xbiout_constf & rst
LOOP1: wrcntr(C0) & 2 8000h+N-3 & ken & xbiout_constf
      yrtr(R0) & dsel & 2 0 & ken & xbiout_constf
```

```
        2 0 & ken & xbain_constf & reg_ewr & e_addr(#20h) & rdxq & xbbport_xi &
            b_addr(#0) & reg_bwr & c_addr(#0) & reg_crd & mul_aen
        wrxq & yinc(C0)(R0) & rddm & xbbport_dmout & b_addr(#1) & reg_bwr &
            c_addr(#1) & reg_crd & mul_ben & e_addr(#20h) & reg_erd &
           alu_ben & d_addr(#20h) & reg_drd & alu_aen
LOOP2:  dccntr(C0) & rdxq & xbbport_xi & b_addr(#0) & reg_bwr & c_addr(#0) &
        reg_crd & mul_aen & sadd
        wrxq & yinc(C0)(R0) & rddm & b_addr(#1) & xbbport_dmout & reg_bwr &
            c_addr(#1) & reg_crd & mul_ben
        jda(sign) LOOP2 & ken & xbiout_constf & moen & aoen & aout_to_in &
            aluout_bufen & alu_ben & d_addr(#20h) & reg_drd & alu_aen &
            a_addr(#20h) & reg_awr
        cont & sadd
        cont
        moen & aoen & aout_to_in & aluout_bufen & alu_ben & d_addr(#20h) &
          reg_drd & alu_aen & a_addr(#20h) & reg_awr
        sadd
        cont
        dccntr(C1) & aoen & xbyout_aluspout
        jda(sign) LOOP1 & ken & xbiout_constf & wryq
        wrcntr(C1) & 2 K*M-cell_no*M-1 & ken & xbiout_constf
LOOP3:  dccntr(C1)
        jda(sign) end1 & ken & xbiout_constf
        rdyq & xbyout_yi
        jda(unconditional) LOOP3 & ken & xbiout_constf & wryq
end1: cont
```

Performance results:

In the sample matrix multiplication of A(3x2) and B(2x3), total number of clock cycles required to execute the program is equal to 70. The total number of floating point multiplications are 9*2 = 18 and 9 floating point additions. Since the matrices are small, the most of the cycles are wasted in initial and final skews in the operation of the cells and finally for transferring the results to the IFU. Therefore the computation power of the SASP is not fully utilized by the multiplication of smaller matrices.

In another example, for the multiplication of matrix A(20x20) and B(20x3), the simulator takes 1525 clock cycles to execute the program. Here it does (20*3)*20=1200 floating point multiplications and (20*3)*19=1140 floating point additions. Thus for a 10 MHz clock rate 15 Mflops rate is achieved using 3 cells.

5.3 The convolution of two sequences :

Let us consider the convolution of two sequences x(n) and h(n),

$$Y(n) = \sum_{k=-\infty}^{\infty} x(k) * h(n-k)$$

Let

$$x(n) = 0, \qquad n<0$$

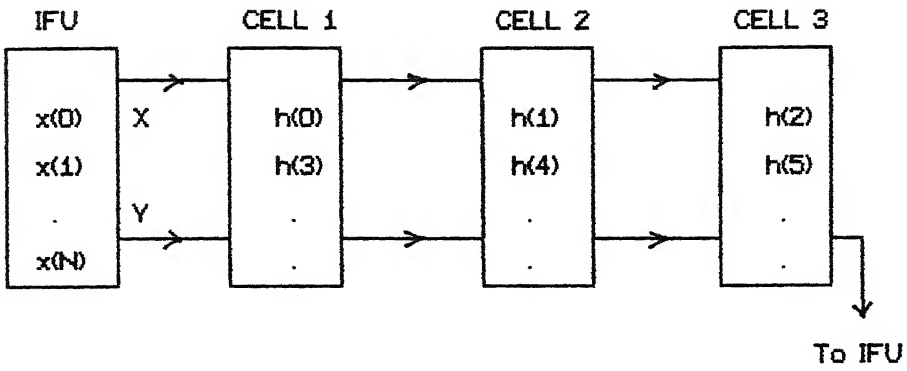
$$h(n) = 0, \qquad n<0$$

then

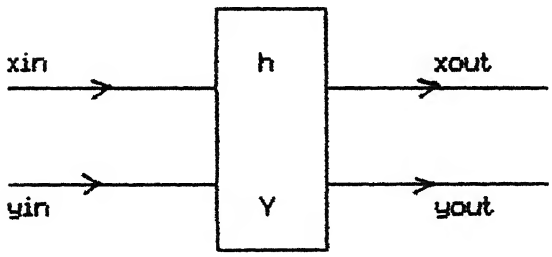
$$Y(n) = \sum_{k=o}^{n} x(k) * h(n-k) \qquad 0 \leq n < N$$

To evaluate the output sequence, the input sequence x(n) is kept in the X memory and weights h(n) are kept in the cell array as shown in figure 5.2. The initial results are kept in the YA memory. In this case these are all zeros.

To execute this algorithm the interface unit reads initial results from the YA memory and the input data from the X memory and writes these data into the Y queue and the X queue of the cell 1 , respectively. A cell n reads the X data from its X queue and multiplies it with the first weight from its data memory. This multiplication result is added to the partial result from the Y queue (written by cell n-1). The result of the addition is written to the Y queue of the cell n+1. The last cell (cell N) writes the partial results into the YB memory. The IFU again reads the partial results from the Yb memory and also starts reading the X data from the first location in the X memory (By clearing the X counter) and writes the read data into the Y queue and X queue of cell 1 respectively. In this second phase the cell n uses next (second) weight from the data memory to multiply the X



The cell specifications



xout = xin

Y = yin + h * xin

yout = Y

Fig. 5.2 Convolution algorithm.

data. This continues till the last weight in the cell array is used. Thus at the end , final results are stored in the YB memory.

The microprograms for the IFU and cell 1 are given below. The last instruction for the last cell (Here it is cell 3) should be 'cont' instead of 'jpcnf' to indicate 'program termination' to other cells.

```
; PROGRAM FOR THE IFU , TO CONVOLVE TWO SEQUENCES.
N       equ     10
no_of_coeff  equ 2      ;Number of weights stored per cell.
cell_N  equ     3

        wrcntr(c0) & 2 8000h+N-3 & ken & clrx & clryb & clrya
        rdx & yrtr(r0) & dsel & 2 8000h-2 & ken
loop:   dccntr(c0) & wrxq & rdya
        jda(sign) loop & ken & rdx & wryq
        wrxq
        wrcntr(c1) & 2 no_of_coeff-2 & ken & rst
loop1:  dccntr(c1) & yinc(c0)(r0)
        jda(sign) end1 & ken
        rtd(r0) & dstb
        wrcntr(c2) & den
next1:  wrcntr(c0) & 2 8000h+cell_N-2 & ken
next:   dccntr(c0) & rdyb
        jda(sign) next & ken & wryq
        dccntr(c2)
        jda(sign) next1 & ken
        wrcntr(c0) & 2 8000h+N-3 & ken & clrx
        rdx
loop2:  dccntr(c0) & wrxq & rdyb
        jda(sign) loop2 & ken & rdx & wryq
        wrxq
        jda(unconditional) loop1 & ken
end1:   jpcnf
```

Since a cell has two weights (coefficients) in its data memory , the X input data has to be routed twice through the array. In the first routing , first three weights from first location of data memories of the three cells are used in the convolution process and the partial results are written into the YB memory. In the

second routing the next three weights are used and final results are written into the YB memory. Thus same cell is used twice , reducing the requirement of more number of cells.

```
;PROGRAM FOR A CELL  to convolve floatiing point sequences. Use of feedback
;method to simulate multiple cells using one cell.
cell_no   equ    1
N    equ   10
no_of_coeff   equ   2
cell_N   equ  3

        wrcntr(c1) & rst & 2 no_of_coeff-2 & ken & xbiout_constf
        yrtr(r0) & dsel & 2 0 & ken & xbiout_constf
        yrtr(r1) & dsel & 2 8000h-2 & ken & xbiout_constf
start:  wrcntr(c0) & 2 cell_no-2 & ken & xbiout_constf
again:  dccntr(c0)
        jda(sign) next & ken & xbiout_constf
        rdyq & xbyout_yi
        jdr(unconditional) again & ken & xbiout_constf & wryq
next:   wrcntr(c0) & 2 8000h+N/2-5 & ken & xbiout_constf & rdxq &
             xbbport_xi & b_addr(#0) & reg_bwr & c_addr(#0) & reg_crd & mul_aen
        yrtr(r0) & wrxq & rddm & b_addr(#1) & xbbport_dmout & reg_bwr &
            c_addr(#1) & reg_crd & mul_ben
        rdxq & xbbport_xi & b_addr(#0) & reg_bwr & c_addr(#0) & reg_crd &
            mul_aen
        wrxq & yrtr(r0) & rddm & b_addr(#1) & xbbport_dmout &
          reg_bwr & c_addr(#1) & reg_crd & mul_ben
        rdyq & xbain_yi & alu_ben &  moen & a_addr(#3) & reg_awr &
           d_addr(#3) & reg_drd & alu_aen & rdxq & xbbport_xi &
           b_addr(#0) & reg_bwr & c_addr(#0) & reg_crd & mul_aen
        sadd & wrxq & yrtr(r0) & rddm & b_addr(#1) & xbbport_dmout &
          reg_bwr & c_addr(#1) & reg_crd & mul_ben
        rdyq & xbain_yi & alu_ben &  moen & a_addr(#3) &
          reg_awr & d_addr(#3) & reg_drd & alu_aen & rdxq & xbbport_xi &
          b_addr(#0) & reg_bwr & c_addr(#0) & reg_crd & mul_aen
        aoen & xbyout_aluspout & sadd & wrxq & yrtr(r0) & rddm &
        b_addr(#1) & xbbport_dmout & reg_bwr & c_addr(#1) &
           reg_crd & mul_ben
loop:   wryq & rdyq & xbain_yi & alu_ben &  moen & a_addr(#3) & reg_awr &
        d_addr(#3) & reg_drd & alu_aen & rdxq & xbbport_xi &
             b_addr(#0) & reg_bwr & c_addr(#0) & reg_crd & mul_aen
        sadd & wrxq & yrtr(r0) & rddm & b_addr(#1) & xbbport_dmout &
           reg_bwr & c_addr(#1) & reg_crd & mul_ben & aoen & xbyout_aluspout
        dccntr(c0) & wryq & rdyq & xbain_yi & alu_ben &  moen & a_addr(#3) &
          reg_awr & d_addr(#3) & reg_drd & alu_aen & rdxq & xbbport_xi &
          b_addr(#0) & reg_bwr & c_addr(#0) & reg_crd & mul_aen
        jda(sign) loop & ken & xbiout_constf & aoen & xbyout_aluspout &
           sadd & wrxq & yrtr(r0) & rddm & b_addr(#1) & xbbport_dmout &
         reg_bwr & c_addr(#1) & reg_crd & mul_ben
```

```
        wryq & rdyq & xbain_yi & alu_ben &  moen & a_addr(#3) & reg_awr &
             d_addr(#3) & reg_drd & alu_aen & rdxq & xbbport_xi &
             b_addr(#0) & reg_bwr & c_addr(#0) & reg_crd & mul_aen
        sadd & wrxq & yrtr(r0) & rddm & b_addr(#1) & xbbport_dmout &
           reg_bwr & c_addr(#1) & reg_crd & mul_ben & aoen & xbyout_aluspout
        wryq & rdyq & xbain_yi & alu_ben &  moen & a_addr(#3) &
           reg_awr & d_addr(#3) & reg_drd & alu_aen & rdxq & xbbport_xi &
           b_addr(#0) & reg_bwr & c_addr(#0) & reg_crd & mul_aen
        aoen & xbyout_aluspout & sadd & wrxq & yrtr(r0) & rddm & b_addr(#1) &
        xbbport_dmout & reg_bwr & c_addr(#1) & reg_crd & mul_ben
        wryq & rdyq & xbain_yi & alu_ben &  moen & a_addr(#3) & reg_awr &
        d_addr(#3) & reg_drd & alu_aen
        sadd  & aoen & xbyout_aluspout
        wryq
        aoen & xbyout_aluspout
        wryq & moen & xbyout_mresult
        dccntr(c1) & wryq & yinc(c1)(r1)
        jda(sign) end2 & ken & xbiout_constf
        rtd(r1) & dstb
        wrcntr(c2) & den
next1:  wrcntr(c0) & 2 8000h+cell_N-2 & ken & xbiout_constf
again1: dccntr(c0) & rdyq & xbyout_yi
        jda(sign) again1 & ken & xbiout_constf & wryq
        dccntr(c2)
        jda(sign) next1 & ken & xbiout_constf
        jda(unconditional) start & ken & xbiout_constf & yinc(c0)(r0)
end2:   jpcnf
```

To see the results the YB memory should be read using 'rdyb' simulator command.

The table 5.1 depicts the operations done by the IFU and cell array on the various data items during the execution of the convolution algorithm. A multiplier in a cell uses two registers 'mul_A' and 'mul_B' and an ALU uses two registers 'alu_A' and 'alu_B'. Result of multiplication is stored in 'mresult' and output of ALU is stored in 'aresult'.

Performance results :

1) For a system with 3 cell, each having 2 weights (Total 6 weights, h(n)) and with 100 input data (x(n) , N =100) , the SASP simulator takes 465 clock cycles to execute above convolution algorithm. In this convolution , number of floating point

Table 5.1

Cycle	IFU (cell 0)	Cell 1	Cell 2	cell 3
1	–	–	–	–
2	Read X memory	–	–	–
3	Write X queue; Read YA memory	–	–	–
4	Read X memory; Write Y queue	–	–	–
5	Same as cycle 3	–	–	–
6	4	–	–	–
7	3	mul-A = x_0 (Read X queue)	wait (for reading Y queue)	Wait (for reading Y queue)
8	4	Write XQ ; mul-B = h_0	wait	wait
9	3	mul-A = x_1 (Read XQ)	wait	wait
10	4	Write XQ ; mul-B = h_0	wait	wait
11	3	mul-A = x_2 (Read XQ); Alu-B = $Y_0 = 0$ (Read YQ); alu-A = mresult = $x_0 \times h_0$.	wait	wait
12	4	Add instruction to ALU; write XQ ; mul-B = h_0.	wait	wait
13	3	alu-B = $Y_1 = 0$ (Read YQ); alu-A = mresult = $x_1 \times h_0$; mul-A = x_3 (Read XQ).	wait	wait
14	4	yout = aresult = $Y_0 + x_0 \times h_0$ Write XQ ; mul-B = h_0.	wait	wait
15	3	Write YQ ; mul-A = x_4 (Read XQ) alu-B = $Y_2 = 0$ (Read YQ) alu-A = mresult = $h_0 \times x_2$.	wait	wait
16	4	Write XQ ; mul-B = h_0 ; yout = aresult = $Y_1 + h_0 \times x_1$.	yout = Read YQ = $x_0 \times h_0$	wait
17	3	write YQ ; mul-A = x_5 (read XQ) alu-B = Y_3 (read YQ) alu-A = mresult = $h_0 \times x_3$	write YQ	wait
18	4	Write XQ ; mul-B = h_0 ; yout = aresult = $Y_2 + h_0 \times x_2$	–	yout = Read YQ = $x_0 \times h_0$
19	3	Operations same as cycle 15 of cell 1	–	write YQ (write YB)
20	4	16	mul-A = x_0 (Read XQ)	–
21	3	17	Write XQ ; mul-B = h_1	–
22	4	18	Operations same as cycle 9 of cell 1	wait (for reading YQ)
23	3	15	10	wait
24	4	16	11	wait
25	3	17	12	wait

Cycle	IFU	Cell 1	Cell 2	Cell 3
26	4	18	13	wait
27	3	15	14	wait
28	4	16	15	wait
29	3	17	16	yout = read YQ = x1 x h0 + x0 x h1
30	4	18	17	write YQ (write YB)
31	3	15	18	—
32	4	16	15	—
33	3	17	16	mul-A = x0 (Read XQ)
34	4	18	17	mul-B = h2
35	3	15	18	Operations same as cycle 9 of cell 1
36 ⋮	4 ⋮	16 ⋮	15 ⋮	10 ⋮

multiplications is approximately equal to 100*6=600 and number of floating point additions is equal to 100*5=500. Thus for a clock rate of 10 Mhz , 24 Mflops computation rate is achieved.

2) If instead of two weights only one weight per cell is used , then the total floating point operations are 100*3+100*2=500 . The simulator takes 240 clock cycles to execute this program. This gives 21 Mflops computation rate.

5.4 CONCLUSIONS:

The simulator is a useful tool to develop the programs and to evaluate the performance of the SASP array using different array sizes. The programs developed give the expected results. It shows that the simulator works perfectly and the facilities provided are adequate.

6. THE META-ASSEMBLER

6.0 INTRODUCTION :

A two_phase meta-assembler is developed . The software is designed to assemble microprograms for several different microprogrammable processor architectures.

A conventional assembler is a dedicated piece of software because , it only recognizes the symbols, which define the instructions of one particular machine and word size. The main feature of microassemblers which distinguishes them from other assemblers is the redefinable multiple-field format of the object code.

A microprogram segment must specify many things: sequence of the microprogram control flow, control codes for ALU and other chips like, address generator, register addresses, timings and enabling conditions for latches and switches, constants of a very few to many bits in lengths for comparison, preloading or masking. The control code groups may be bit patterns for direct control of gates, or they may be encoded functions.

The typical microinstruction, then, is a bit pattern of several fields; each of the fields may have different lengths in bits. For specifying the content of each field an assembly language requires multiple assignments. This microassembler allows the assignments of all the field bit patterns as a group.

The typical line of a microprogram therefore, differs from a conventional one word computer instruction line in that, it has multiple "opcodes". Furthermore, different opcode patterns might call for different field groupings in successive

microinstructions. A jump instruction, for example, might call for only two fields ; the field specifying the jump function and a long field giving a jump address within the microprogram memory. On the other hand , an ALU operation might require several short fields containing codes giving the first and second operand source locations within the register set, the codes for the ALU operation performed, the destination of the result , and bits to control the handling of carries and condition codes.

The microassembler developed in this thesis can be redefined to translate the source code for many different machines. Because of this redefinability this assembler is also called as meta-assembler. This is achieved by splitting the assembly into two distinct phases.

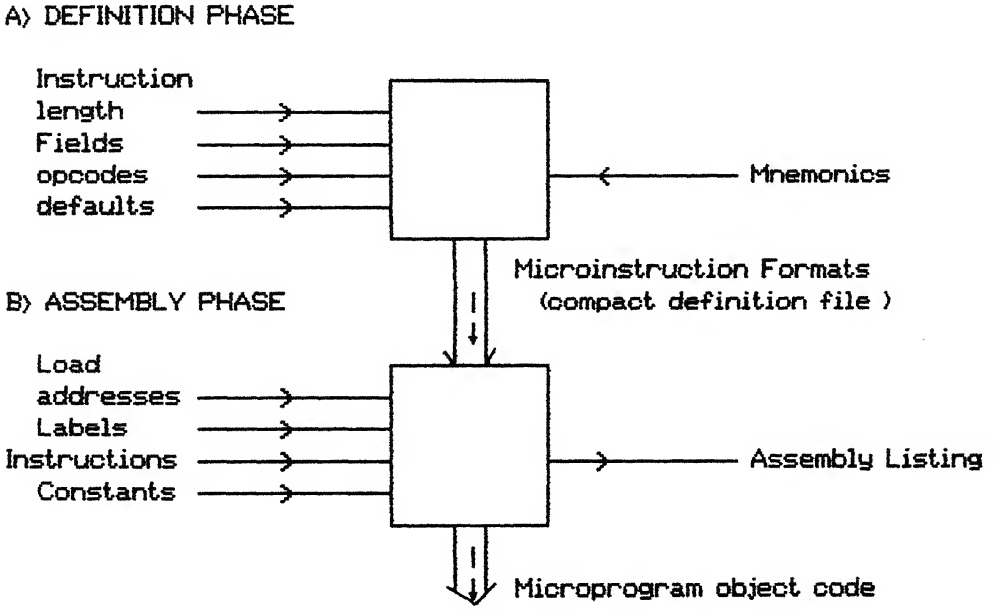


Fig. 6.1 Phases of the meta-assembler.

6.1 DEFINITION PHASE:

The first phase or definition phase processes a file containing a definition of the field format of an instruction word in the target machine. The word is defined by its name, position in the word, width, default value, a set of mnemonic tags to be associated with opcodes and association of these tags with appropriate formats and field values, combination of values which are not legal, branch instruction which need branch address to be given in the data field and opcode bit notations and associated symbols used. This information is translated into an intermediate format, and stored in a file.

6.2 ASSEMBLY PHASE:

The second, or assembly phase is more like assembly language processing of a fixed-architecture computer. The basic task is to scan lines of input symbols, translating mnemonic codes , expressions, and controls into sequences of binary microinstructions. Fields specified in the definition phase are pulled together, aligned, and stuffed into a word called microinstruction. Thus output of this phase is the microprogram object code.

6.3 ASSEMBLY LINE:

A typical assembly line should start with an alphanumeric label and be followed by an instruction-flow opcode (sequencer instruction). Blanks and tabs are allowed as separators among the items in the lines. Because of the generally complex structure of the microinstruction format, a variable number of items

follow. One microinstruction is allowed to spill over several lines. The assembler allows for missing opcodes and values when a programmer desires a default quantity to be inserted in the corresponding field. Comments are allowed , starting with a semicolon, and ending with new line character.

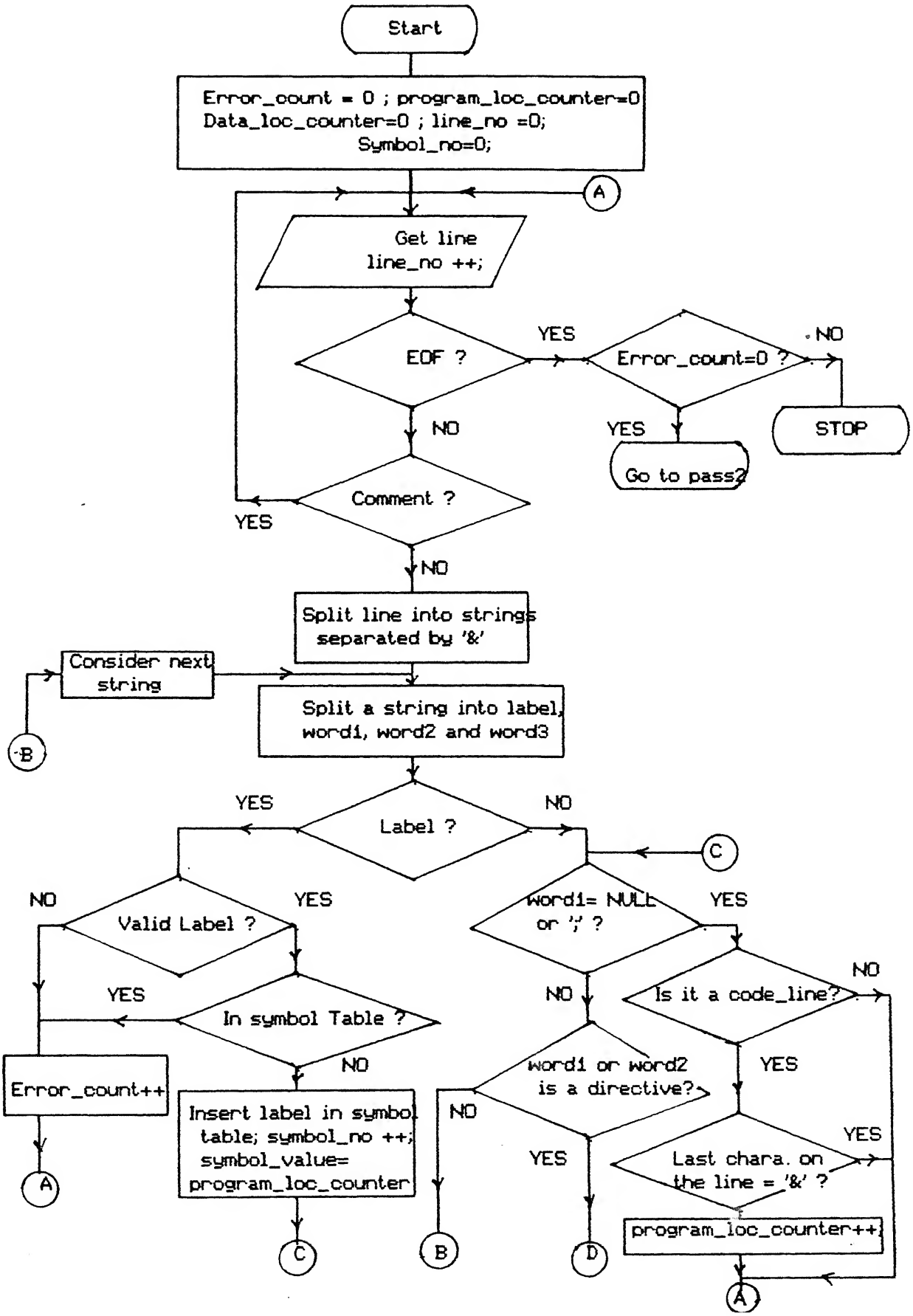
6.4 IMPLEMENTATION :

Both phases of the assembler are implemented in C language. The original goal of the developing this software was to provide software support for the linear systolic array processor, which is under development. This has been achieved. The only short coming of this meta-assembler is the absence of macroprocessing facility, which can be added.

The two phase assembly has one more advantage, that is the intermediate (compact) definition file can be used by a general purpose symbolic debugger to provide symbol definitions for data entry and display while debugging.

The flowchart of the program for assembly processing phase (phase 2) is as shown in fig. 6.2.

Fig. 6-1a Continued.



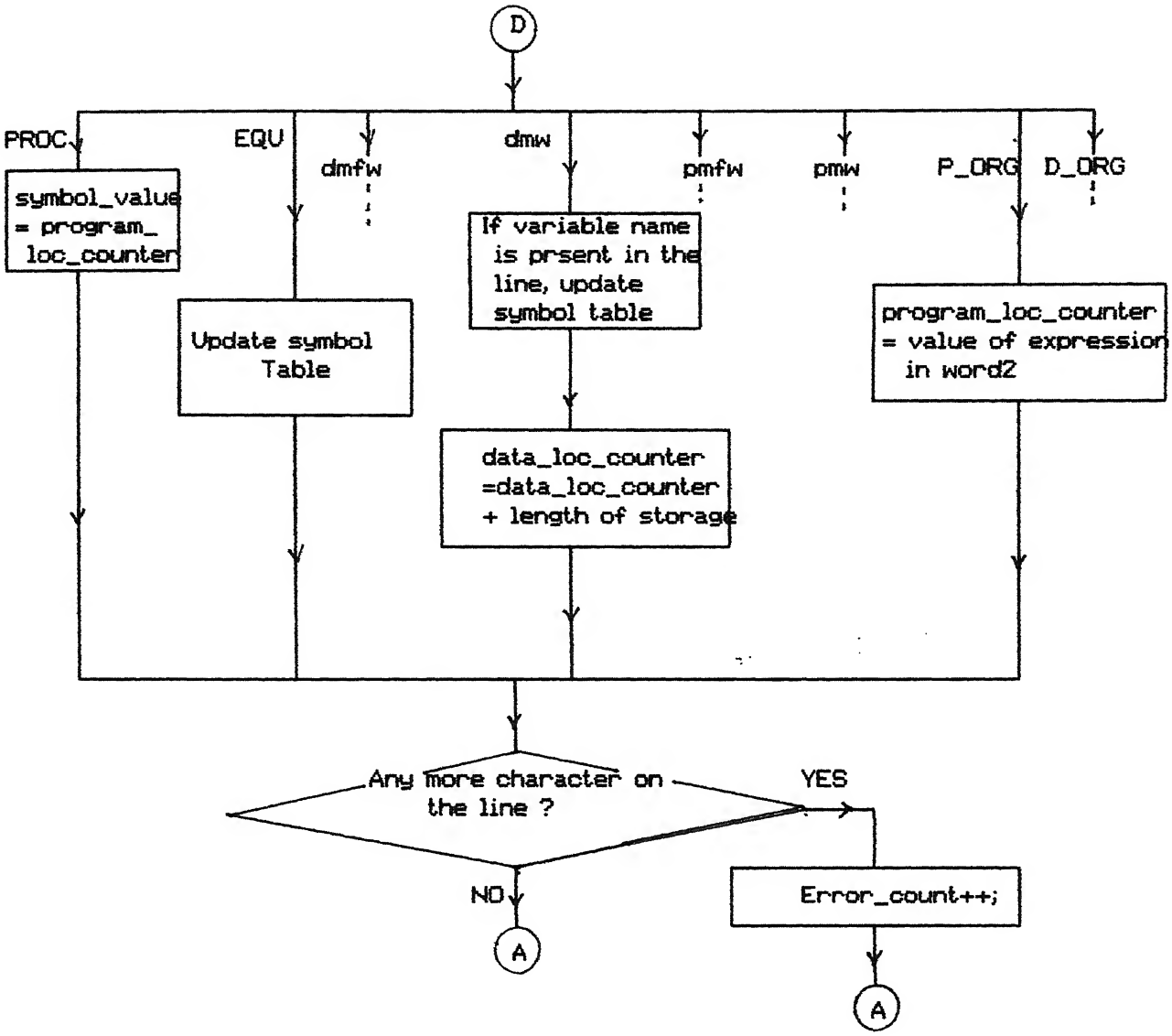


Fig. 6.1 Flowchart for PASS1 of the ASSEMBLY PHASE .

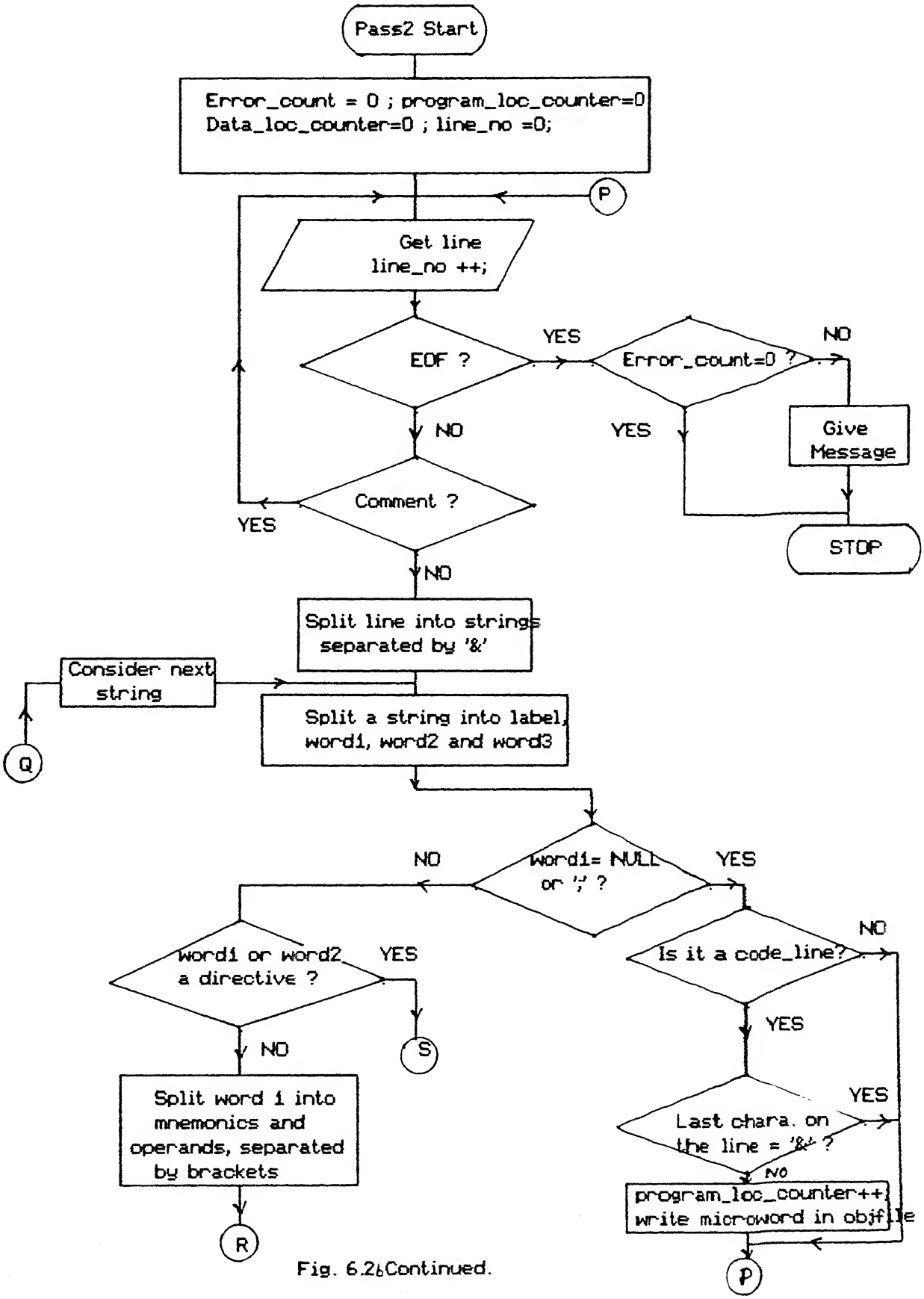


Fig. 6.2b Continued.

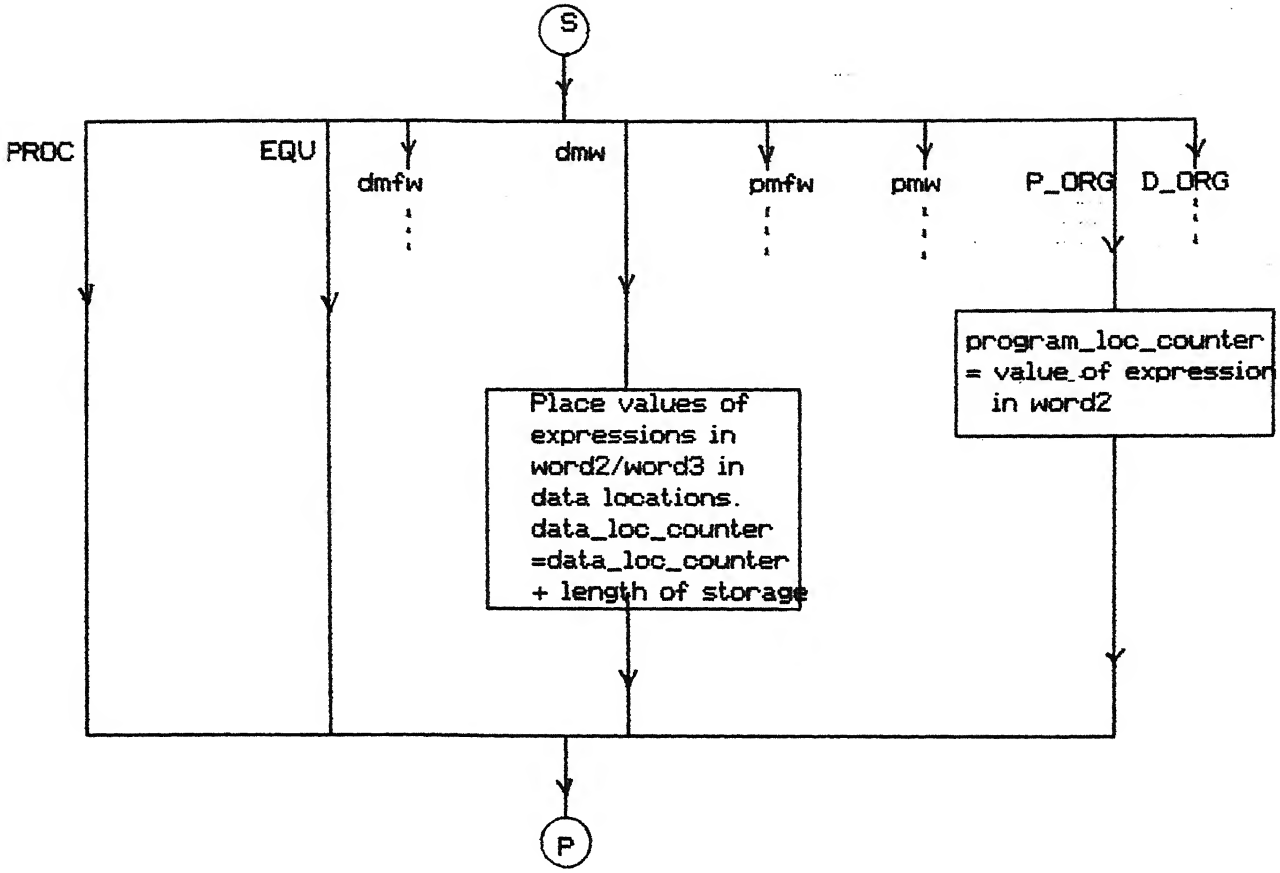


Fig. 6.2bContinued.

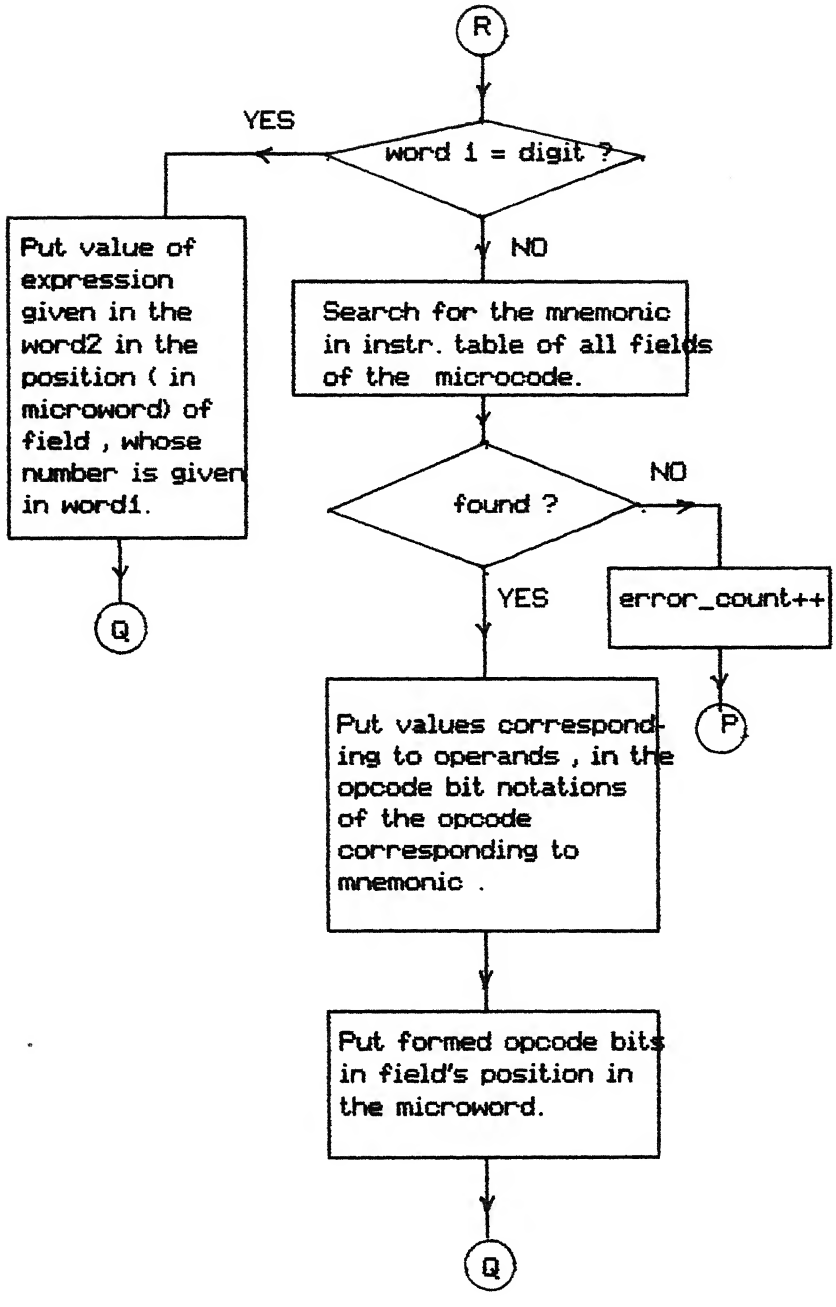


Fig. 6.2bFlowchart for PASS2 of the ASSEMBLY PHASE

7. CONCLUSIONS

CONCLUSIONS:

A systolic array signal processor (SASP) has been described. It is a linear array of microprogrammed cell units connected to an external host through an interface unit.

A simulator for the SASP is developed, which can be used to develop and debug the programs for the SASP system. The simulator is a useful tool for evaluating the performance of the system. It provides a tool for testing programs to be run on the SASP hardware. It also provides the facility of running the program under user control and estimates the run time in number of cycles.

A meta-assembler is developed, which is used for assembling the programs for the SASP system. The meta-assembler is generalized and redefinable, so that it can be used for any microprogrammed system or a microprocessor with fixed instruction opcode size. The meta-assembler has been thoroughly tested on sample programs.

The programs developed for the SASP give the expected results. And it shows that system simulator works perfectly and the facilities provided by the simulator are adequate. The simulator has been tested using the matrix multiplication and the convolution algorithms for few sample examples.

SUGGESTIONS FOR THE FUTURE WORK :

1) More algorithms can be developed for a linear systolic array using the simulator. e.g. FFT , AR filtering , Matrix operations like finding eigen values, etc.

This will also help to evaluate the performance of the SASP system.

2) The simulator blocks (sequencer , address generator and floating point units) can be used to simulate other microprogrammed architectures also.

3) The modularity of the simulator program permits to modify it to match the further developments in the hardware design, so that the simulator can be used for developing and debugging programs for the target system.

4) The meta-assembler is a very useful tool , which can be used for any microprogrammed architecture. An addition of MACRO facility to the meta-assembler will make it more powerful and complete .

5) An optimizing compiler can be developed , which will hide the low - level details of the system and allow the user to concentrate more on the parallelism at the array level. The W2 language developed for the CMU Warp processor [Anna87] is an example of such software support.

REFERENCES

Anna87 M . Annaratone , E . Arnould, ' The warp computer Architecture, Implementaion, and Performance', IEEE Trans. on computers, c-36, No.12, Dec 1987.

HTKung82 H.T.Kung, 'Why Systolic Architectures ?', Computer Magazine, Jan 1982.

Hwang84 K. Hwang and F.A.Briggs, 'Computer Architecture and Parallel Processing', Mc Graw Hill, 1984.

Kesh87 M.R. Kesheorey , 'Speech processor architectures and Simulator of TMS 32010' , M.Tech Thesis, IIT , Kanpur , 1987.

Nemm88 S.S.Nemawarkar 'SASP : A Systolic Array signal Processor', M.Tech Thesis, IIT, Kanpur, 1988.

Powers78 V.M.Powers, J.H.Hernandez, 'Microprogram Assembler for Bit-slice Microprocessors', Computer Magazine, July 1978.

SAM89 Samit Choudhary, 'Implementation of an Interface Processor and Design of Algorithms for SASP', M.Tech Thesis, 1989.

Scholes84 N.F.Scholes, P.K.Lala, 'Redefinable cross Assembler for horizontally microprogrammable Processors', Microprocessors and microsystems , Vol. 8, no. 4 , May 1984.

SYKung87 S.Y.Kung, 'VLSI Array Processors', Prentice Hall Information and System Sciences Series, 1987.

Swartz87 E.E.Swartzlander, Jr., 'Systolic Signal Processing Systems', Marcel Dekker, Inc. 1987.

Usman89 Mohd. Usman, 'Design of SASP: A Systolic Array Signal Processor', M.Tech Thesis, IIT, Kanpur, 1989.

MANUALS

1) DSP Product databook, Analog devices, 1987.

2) ADSP-2100 Cross-Software Manual, Analog devices.

3) CMOS BiCMOS Data Book , Cypress Semiconductor.

APPENDIX A

MANUAL FOR META-ASSEMBLER

A.0 INTRODUCTION :

There are two phases of the assembly. The first phase processes a file containing a definition of the format of an instruction word in the target machine. This information is translated into an intermediate format, and held in a file on backing store.

The second or assembly phase reads the translated definition file from backing store and uses the symbols contained within it to process the programmer's source code. The output from the second phase is a text file with one assembled microinstruction per line. The output is in a simple format coded in binary or hexadecimal.

A.1 PHASE 1 - DEFINITION PROCESSING

The definition file can be written in any standard text editor. The upper case and lower case letters are not distinguished.

This phase is best illustrated by a simple example. In the example reserved words appear in upper case to improve readability (not because it is a requirement) and a semicolon introduces a comment which is terminated by the end of the line. There are no layout restrictions.

A definition file starts with,

TITLE demonstration

The TITLE directive has no effect on the translation and is included as an aid to quick program identification. If TITLE is omitted translation is aborted early on. Thus little time is wasted on a probably erroneous file.

The output from the definition translator can be directed to a file with a particular name. The directive,

DEFN_FILE "test.dat"

causes the output to be sent to the file with the name within the quotation marks. This file then will be the input to the phase 2 translator.

Let us consider an example of a system consists of a MAC, an address generator and a sequencer. The word is 45 bits wide and is divided into the fields as shown in fig. A.1. There are total 8 fields.

The next step is to give the word size of the target processor. Here it is 45 bits. The input is

WORD WIDTH 45 BITS

The reserved words WIDTH and BITS are optional.

The next step in the definition file is to give the program memory location width. Generally it is equal to the word width. The input is

PROGRAM_LOC_WIDTH 45 BITS

BITS is optional.

The next line should be

DATA_LOC_WIDTH 16 BITS

BITS is optional. This indicates the data memory location width. In this

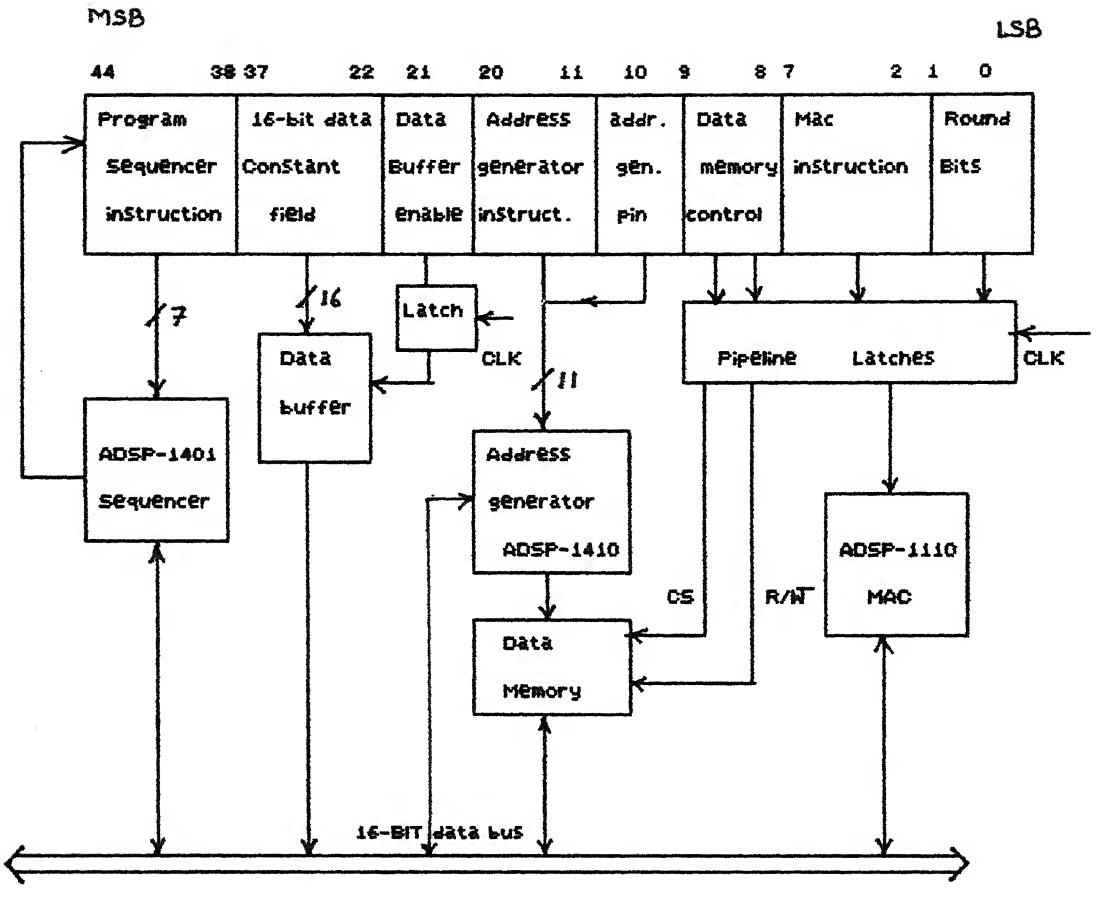


Fig. A.1 Example system.

example, it is 16 bits.

The fields within the microinstruction word are defined next. The first field in this example can be defined as,

#sequencer : FIELD 38-44 , WIDTH 7 , DEFAULT cont

The name of the field is 'sequencer'. It starts at bit 38 and ends at bit 44. The dash '-' between 38 and 44 is essential and no space is allowed in between. The clause WIDTH 7 is useful, because, it is checked against the width calculated from the start and end bit positions of the field, which results in an error, if disagreement is detected. The DEFAULT clause is used to assign a default value to a field for which no value is specified in phase 2.(assembly phase). It should be given as DEFAULT mnemonic and no integer or binary value is accepted instead of mnemonic. This mnemonic should be present in the VALUES clause, given at the end of a field declaration.

Within a field or microinstruction, the bits are numbered as ascending order of significance from right to left. The lower-numbered bit has the least significance.

The next clause in a field is

```
OPCODE_BIT_NOTATIONS [
        kk (
;The values and symbols should be written on the next line.
        00      flag
        01      carry
        10      equal
        11      nocondition
        )
        cc (
          c0-c3
        )

        ]
```

This clause is optional and can be omitted. Here kk and cc are bit notations.

Thus this clause gives information about the bit notations, used in the opcodes, defined in the clause VALUES. It gives the bit notations and the values taken by them for different symbols. These symbols are then used in the operand field of an instruction in assembly phase.

If a bit notation cc represents registers C0-C3, then it is not necessary to write every register's name and value for bit notation, just C0-C3 is sufficient. The only restriction is that, the bit notation letters and register's name should start with same alphabet.

In the bit notation some characters are not allowed, they are ';', ',', ':', ')', '(', '[', ']' and numbers '0'-'9'. All the characters in the bit notation must be same. The length of bit notation string should be less than 32 characters. The length of symbol used should not exceed 32 characters. Number of symbols used for a bit notation should not exceed 32. It is also limited by the length of the bit notation. e.g. For kk (length 2) maximum 2power2=4 symbols should be present, since there are only 4 combinations of bit notation values possible. Here, in above example, for the bit notation 'kk', there are 4 symbols 'carry', 'equal', 'nocondition' and 'flag'. The clause starts with '[' and ends with ']'. Each bit notation's symbols and associated bit notation values are embedded in brackets '(' and ')'.

The next clause in the field is

```
INSTR_NOT_AVAILABLE [
        branch(sign)(c0)
        branch(sign)(c1)
        ;..
        ]
```

This clause is also optional. In some mnemonics some operand (condition)combinations are illegal and no opcode is available for such instructions. Then such instructions (with operands) should be written in this

clause. The symbols in the brackets are operands/conditions. If an instruction present in this clause appears in the source code (phase 2) assembler gives an error. The next clause in the field is

```
BRANCH_INSTR_WHICH_NEED_DATA [
        ABSOLUTE (
            jda
            ;..
            )
        RELATIVE (
            jdr
            ;..
            )
            ]
```

This clause is also optional. This is mainly useful for sequencer field.In it, the mnemonics for branch instructions, which need jump address to be in the data field, are written without their operands/conditions. In phase 2, for these instructions the value corresponding to the label or expression written next to instruction is filled in the 'data' (constant) field. The name of the data (constant) field must be "data" and no other name is allowed.

For mnemonics written under ABSOLUTE clause, in the phase 2 the absolute address corresponding to label (/expression) given in next word (word2) to the instruction, is put in the 'data' field. And for the mnemonics, written under RELATIVE clause in the phase 2, relative address corresponding to the label/expression is put in the 'data' field.

The next clause in the field is

```
VALUES [
            idle    0010000
            jda     111kk11
            cont    0000000
            ;...
            ]
```

In this clause, mnemonics and corresponding opcodes for the field ar

written. The opcode width should match with as given in the field width. This clause is compulsory, since every field has some default given in terms of a mnemonic. Therefore atleast one mnemonic should be present in this clause.

The other fields for the given system are defined in a similar way. The definition file contents look like as follows.

```
TITLE TRIAL
DEFN_FILE "TEST.DAT"
WORD WIDTH 45 BITS
DATA_LOC_WIDTH 16
PROGRAM_LOC_WIDTH 45
;serial no. 1
#sequencer:    FIELD 38-44,WIDTH 7 DEFAULT cont
 OPCODE_BIT_NOTATIONS [
       kk (
            00        unconditional
            01        notflag
            10        flag
            11        sign
            )
       cc (
            C0-C3
            )
       ii (
            I0-I3
            )
       ]
 INSTR_NOT_AVAILABLE [
       branch(sign)(c0)
       branch(sign)(c1)
       branch(sign)(c2)
       branch(sign)(c3)
       ]
 BRANCH_INSTR_WHICH_NEED_DATA [
     ABSOLUTE (
       jsa
       jda
       )
     RELATIVE (
       jsr
       jdr
       )
       ]
 VALUES  [
           ;jump & branch instruction
         jda       111kk11
         jdr       111kk01
         jsr       111kk10
```

```
            branch    100kkcc
            dccntr    01100cc
            enair     0110110
            ;miscellaneous instructions:
            cont      0000000
            idle      0010000
            ;.....
            ]
;serial no.  2
#data: FIELD 22-37,WIDTH 16,DEFAULT zero
 VALUES [
            zero      0000000000000000
            ]
;serial no. 3
#data_enable: FIELD 21-21,WIDTH 1,DEFAULT disable
      values [
            enable    1
            disable   0
            ]
;serial no. 4
#address_generator: FIELD 11-20,WIDTH 10,DEFAULT nop
 OPCODE_BIT_NOTATIONS [
       cc (
            c0-c3
            )
       rrr (
            R0-R7
            )
       rrrr (
            r0-r15
            )
       bb (
            b0-b3
            )
       cc (
            c0-c3
            )
       ii (
            i0-i3
            )
      ; .....
            ]
VALUES [
            yinc      1011ccrrrr
            ysub      11ccbb0rrr
            yrtr      000101rrrr
            yrtb      0011bbrrrr
            yrtc      0010ccrrrr
            itd       00001110ii
            yor       0111bbrrrr
            yxor      0101bbrrrr
            nop       0000000000
            ;.....
            ]
```

```
;serial no. 5
#ad_gen_pin:        FIELD 10-10,WIDTH 1,DEFAULT nodsel
      VALUES [
            dsel      1
            nodsel    0
            ]
;serial no. 6
#data_mem_control: FIELD 8-9, WIDTH 2 DEFAULT norw
      VALUES [
            rd        11
            wr        10
            norw      00
            ]
;serial no. 7
#mac: FIELD 2-7,WIDTH 6,DEFAULT nop
   VALUES
            [
            nop       000000
            x=bus     001000
            ls=bus    010000
            bus=ex    001101
            mul       100000
            muladd    100010
            mulnadd   100011
            bus=ls    011101
            ;.....
            ]
;serial no. 8
#mac_rnd_pins: FIELD 0-1 WIDTH 2 DEFAULT nornd
      VALUES [
            rnd14     01
            rnd15     10
            nornd     00
            ]
```

The microinstruction format has now been completely defined. If no errors are detected during processing of the definition file, this microinstruction format is written into a compact file named "test.dat".(The name is given in the DEFN_FILE clause).

A.1.1 PROCESSING OF THE USER FILE:

To process the user file, user should give following command.

DEF FILENAME (or)

or

DEF (cr)

def_file. FILENAME (cr)

The DEF program processes the user definition file and generates compact definition file, which is then read by the assembler (phase 2) to assemble the source code.

A.2 PHASE 2 : ASSEMBLY PHASE

The second phase is a conventional two-pass assembly. First the compact definition file is read and then it starts processing the programmer's source code. The output files of the assembler are,

-an ASCII file containing the resulting object code.(.obj)

-a list file. (with error listing if present.) (.lst)

-a symbol file. (.sym)

The errors are also displayed onto the standard output device.

A.2.1 Running the assembler:-

To invoke the assembler, the command form is :

MEASM [-switch][-switch] (cr)

The switches are optional and are

-l To create list file.

-b To create binary coded object file.

(Hex coded object file is by default)

After pressing (cr) the assembler asks for src file name and definition file name.

A.2.2 Language Conventions:-

This section covers the language conventions (symbols and constants), used in the source code file. The assembler does not distinguish between upper case and lower case letters.

A.2.2.1 Symbol Conventions

All symbol names in the source code file must be unique. Symbol names should be of length less than 32 characters. A valid symbol starts with a letter followed by any mix of letters, numbers or underscores. User defined symbols cannot be the same as Assembler key words. The keywords are

include, equ, pmw, pmfw, dmw, dmfw, proc, endp, p_org, d_org, define and dup.

A.2.2.2 Constants

The assembler accepts binary, octal, hexadecimal and decimal numerical constants. They are postfixed with B or b, O or o, H or h, D or d respectively, with the default being decimal. A hex number must start with a numeric.

e.g. 1234h , 0B123h, 1770o, 11111b, 1234

Symbolic constants are assigned by the 'EQU' directive.

e.g. symbol1 EQU 234h

Symbolic constants can be used anywhere to replace numeric constants.

A.2.3 SOURCE CODE FILE:-

In a source code file, programmer can have several modules and a main program. The modules are declared with,

name of module PROC

and ended with ENDP. The body of the source code and file consists of two sections : the code section and declaration section. The sections can be placed in any order or anywhere in the file.

Comments can be inserted anywhere in a source code file, starting with ';'. The every character after ';' on that line is treated as a part of the comment.

A.2.3.1 INCLUDE directive:

Number of assembly files can be included in the original source file by this declaration. e.g #include test2.asm

It includes the source file test2.asm and this file is assembled first before assemblying the next line of the present source file. Nesting of the include files is limited to 10 levels. But this may cause problem (on PC(DOS O.S.)) of number of open file handles at a time, in case if in config.sys file, the line file=15 (or greater number) is not added. The number of include files that can be included by a source file is unlimited.

A.2.3.2 Declaration Section:-

The declaration section uses assembly directives to declare assembly constants, program memory data (variables) , data memory data (variables) , origin of data and program memory that follows origin command.

For variable declaration, in case of data memory 'dmw' (for integer data) and 'dmfw' (for single precision floating point data) are the assembly directives to be used. Variable name can be given before 'dmw' or 'dmfw' directives. e.g.

```
Buffer    dmw        25,25,455h,334,11110b,2233h,55443
```

dmw 234,6477,636

The data for each location should be separated by ',' and no space is allowed till declaration of last location on that line.

In case of program memory variable, the directive used is 'pmw' or 'pmfw'.

e.g. buffer2 pmw 255h

Each data item is placed in a full word.

To duplicate same data in the memory, 'dup(constant)' directive can be used. The constant represents a number which can be placed in one memory location. A floating point constant cannot be used here. Multiple locations cannot be duplicated.

e.g. 100dup(200h)

200h is duplicated in 100 locations

e.g. 100dup(200h,100h) is invalid.

For 10dup(?), 10 locations will be filled with zeroes.

The starting location of program or data memory area can be declared by P_ORG and D_ORG directives respectively. e.g.

P_ORG 100h

This directive can be used anywhere in the source file. Thus different code segments can have different starting locations.

A.2.3.3 Code Section:-

A microinstruction code statement has the following format.

label: instr1 & instr2 & instr3 & ... &

instrn-1 & instrn

The instructions for each field are separated by '&'. The end of the microinstruction is indicated by the end of line , if the last character on that line

is not '&'. Thus a microinstruction can be several lines long. The instructions in the microinstruction can be written in any order. If a particular field is omitted then default value is written by the assembler at the corresponding position in the microword. If for a field, instead of instruction, defined in the definition file, a numeric expression or a symbol is to be written, then the field's serial number (as in the definition file) should be written before the expression or symbol. In this case value corresponding to the expression is put in the field position.

Arithmetic expression containing symbols and numeric constants can be used throughout the assembly

e.g. buffer+1, buffer*2+3, buffer+buffer2+5

e.g buffer+8 will indicate the address of 9th memory location in the buffer.

The label field is optional, and when it is present, a value corresponding to the address of present program location is assigned to the label symbol.

An instruction may consists of operands, when there are opcode bit notations present in the opcode (given in definition file) of that instruction. The value corresponding to operand symbols is put for the opcode bit notations. The order in which the operands should be written (separated by brackets) depends on the order, in which corresponding bit notations are written in the opcode field. e.g.

```
yinc        1011ccrrrr ,opcode
```

is an instruction for address generator, where yinc is mnemonic and 1011ccrrrr is opcode. cc selects one of the c0-c3 registers. rrrr selects one of the r0-r15 registers.

Thus, for this instruction, there are two operands . Thus the instruction should be written as,

yinc(c0)(r0)

if the operands are c0 and r0

The instruction yinc(r0)(c0) is invalid.

The number of such operands for an instruction is limited to 10. An instruction should be written with no spaces in between.

For branch instructions, which need jump address to be written in the 'data' field , instead of writing the address expression as a separate expression for 'data' field (with 'data' field's serial number, as given in the definition file)', jump address can be written in the form of an expression containing constants and symbols after the branch instruction mnemonic, before the next instruction of other field starts. If the address expression is written in the branch instruction, for relative branching, relative branch address is put in the 'data' field.

e.g. jda(sign) label1 & ....

Here jda(sign) is a branch instruction and label1 indicates the jump location. Thus address corresponding to label1 is put in the 'data' field. This can also be written as,

jda(sign) & 2 label1 & ....

There are two types of branch instructions, 1) which need absolute branch address , 2) which need relative branch address. This should be mentioned in the definition phase.

Once branch address is written in a branch instruction in an microinstruction, then 'data' field of that microinstruction should not be written again, otherwise assembler gives error message

In some machines, there is no separate data field and the branch address data is given in the opcode of the branch instruction itself. In this case the

address is written as an operand of that instruction. In these operands labels(symbols) can be used e g

```
jump        101######### ,opcode and mnemonic
```

Here jump address is given in the operand of the instruction. The bit notation for this address space is ######### means it can take values from #0 to #255 decimal. One can write it as #0 to #0ffh or #0 to #const1 , where const1=0ffh

Thus the instruction may look like jump(#label+2)

A.2.4 EXAMPLE:-

An example program is given. The assembler output file (.obj) is also shown. This program does matrix multiplication on the defined architecture in the definition phase as an example

```
;       c_matrix=a_matrix*b_matrix
        d_org       100h
a_matrix        dmw     12,15,25,78,34,36,28,39,49      ;M*N matrix
b_matrix        dmw     13h,25h,25,37,25,48,69,58,60    ;N*K matrix
c_matrix        dmw     18dup(0)                        ;M*K matrix
                ,For c_matrix each element is of two words.
        N       equ     3
        K       equ     3
        M       equ     3
```

file 1 : define.asm

```
;This program is a sample program to demonstrate use of
;definition/assembly phase of the meta-assembler.

#include        define.asm

            wrcntr(C0) & 2 K-2 & enable         ;initialize r0 -> counter0
            2 M*N & enable & dti(i0)
            itr(i0)(R0)
            wrtc(c0)(r0)                            ;initialize c0
            2 2*M*K & enable & dti(i0)
            itr(i0)(r0)
            wrtc(c2)(r0)                              ;initialize c2
            2 a_matrix & enable & dti(i0)           ; initialize i0
```

d0103	004E
d0104	0022
d0105	0024
d0106	001C
d0107	0027
d0108	0031
d0109	0013
d010A	0025
d010B	0019
d010C	0025
d010D	0019
d010E	0030
d010F	0045
d0110	003A
d0111	003C
d0112	0000
d0113	0000
d0114	0000
d0115	0000
d0116	0000
d0117	0000
d0118	0000
d0119	0000
d011A	0000
d011B	0000
d011C	0000
d011D	0000
d011E	0000
d011F	0000
d0120	0000
d0121	0000
d0122	0000
d0123	0000
p0000	0E0000600000
p0001	00000261E000
p0002	000000100000
p0003	000000040000
p0004	000004A1E000
p0005	000000100000
p0006	000000050000
p0007	00004021E000
p0008	000000100000
p0009	00004261E800
p000A	000000108800
p000B	000044A1F000
p000C	000000111000
p000D	000000E61C00
p000E	000000048800
p000F	0E4000600000
p0010	0E8000600000
p0011	000000200034
p0012	000000000054
p0013	000000000044
p0014	000000160320

```
p0015       000000160B88
p0016       0C8000000000
p0017       1F4000A00000
p0018       1C7FFF200000
p0019       0C4000171274
p001A       000000171274
p001B       000000171264
p001C       1FC007A00000
p001D       1CC0043A0800
p001E       0C0000100000
p001F       000000100000
p0020       1FC008A00000
p0021       1CC003E00000
p0022       0D4000000000
$$
```

Output file: matrix.obj

The assembler also produces a symbol file, which can be used for disassembly of the object file. The first string on a line is the symbol followed by it's value. Every file has some character sequence at the start and end of the file for proper identification, while accessing these output files by the other programs.

```
[[d
A_MATRIX 0100d
B_MATRIX 0109d
C_MATRIX 0112d
LABEL1 000Fp
LABEL2 0010p
LABEL3 0014p
TEMP1 0019p
TEMP2 001Ep
TEMP3 0022p
[[o
```

symbol file: matrix.sym

Note: '[' stands for (esc).

A.2.5 Error Messages:-

The assembler gives following error messages.

1. Syntax error

-for a syntax error

2. ERROR: Number is invalid

-If a numeric constant has invalid characters in it, assembler gives this error.

e.g. 23g0h is invalid, since for a hex constants valid characters are 0-9,a-f.

3. ERROR: Symbol redefined

4. ERROR: Data field is already defined

-For a microinstruction, in which if the sequencer has a branch instruction which need jump address to be in the 'data' field and if a symbol expression is written for the jump address in the sequencer instruction. And again if 'data' field is also assigned some value in the same microinstruction , the assembler gives this error.

5. ERROR: Symbol not found.

6. Error in line.

-For errors other than above this error message is given by the assembler.

7. ERROR: mnemonic for a field written twice in line.

-When two or more instructions are written for single field in a microinstruction, this error is given by the assembler.

APPENDIX B

SIMULATOR MANUAL

B.1 *INTRODUCTION* :

This simulator provides an easy way to verify systolic array system designs without committing to hardware development. As shown in figure B.1 , the simulator reads the architecture description file and the object code files and symbol table files outputted by the meta-assembler An architecture Description file is input to simulator to start a simulator session. The object code files are loaded interactively The Symbol Table files are loaded implicitly when the object files are loaded The definition files are loaded implicitly when the simulator is run.

By reading the Debug symbol Table, the simulator interacts with the user symbolically The user can make references to variables and program labels using symbols defined in the user program, avoiding the need to decode the symbols. The Simulator can disassemble a microinstruction, making full use of symbols defined in the user program

The simulator is interactive. By reading the Architecture Description file it configures itself to match the target system hardware.

Using upload/download files, the user can load data memory (to simulate data generated or loaded by external devices, Host) to the simulator and later upload simulator processed data to a file for subsequent analysis.

B.2 *Architecture Description File* :

A valid architecture file starts with character '@' in the first line. In the

architecture file user can specify architecture of the system. i.e. he can specify length of x queue , y queue, addr queue, and size of data memory, program memory, scratch pad memory for a cell and x, yb , ya memory sizes. He also can specify number of cells in the array. If one of the above parameters is not specified some minimum default value is assumed. The values assigned to these parameters should not be less than the default values, otherwise default value is assumed.

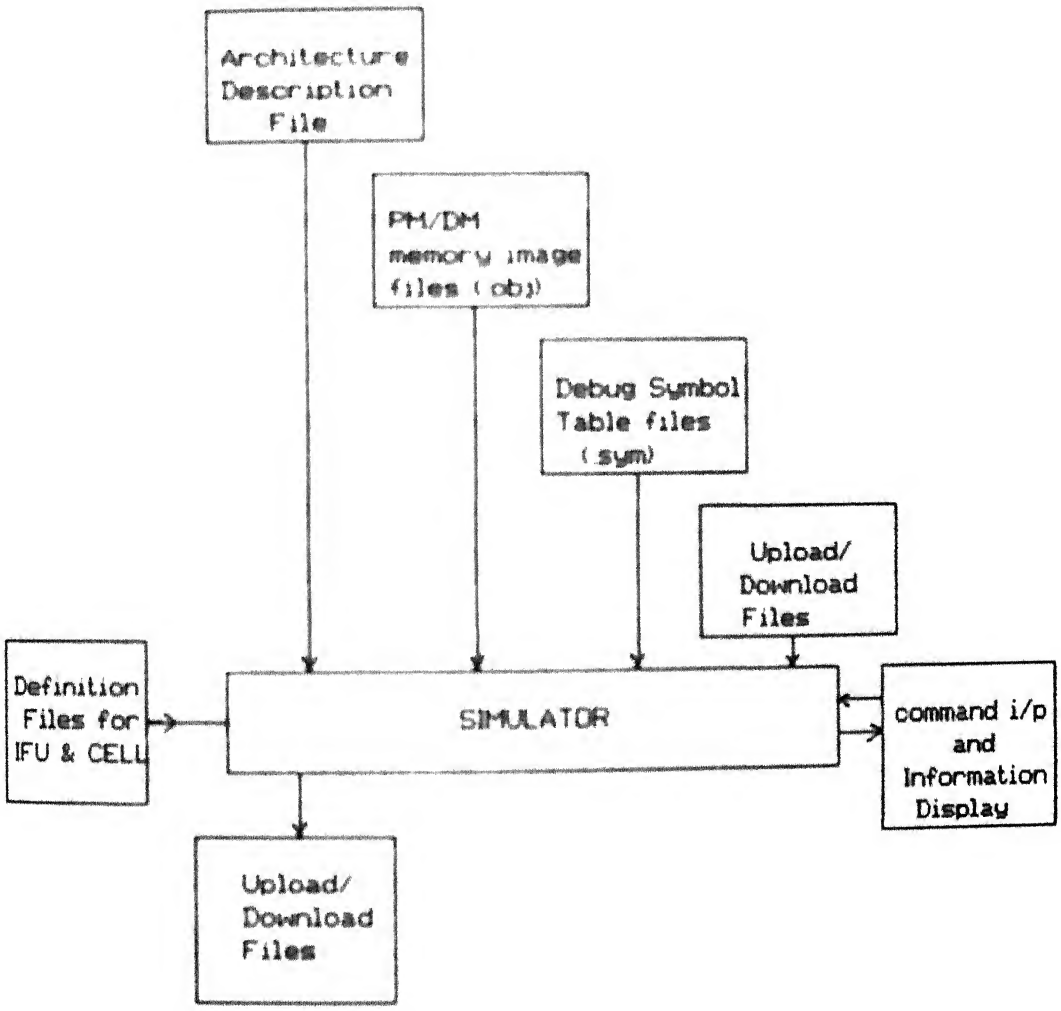


Fig. B.1 Simulator input output files.

The default minimum values are

parameter	size
X memory	1024
Ya memory	1024
data memory	1024
program memory	200
X queue	128
Y queue	128
Addr queue	128
scratch pad	512
register file	512
no of cells	1

An example of architecture file is as follows

```
@
ya              2028
x               07ffh
yb              04ffh
xq              0ffh
yq              512
addrq           256
dmem            2028
scratchpad      600
program_mem     400
no_of_cells     5
@
```

These parameter values can be given in any order.

B.3 Starting the *simulator*

To start the simulator type

SSIM (cr)

Arch_file syst arh

Architecture file tells the simulator how to configure data memory, program memory, queue sizes, number of cells in the array, etc. There can be number of cell units apart from the interface unit (IFU). The commands given at the simulator prompt are meant for the present command cell. When the simulator is started the command cell is the IFU or cell 0. The command cell can be changed by giving command "element(cr)" at the sim) prompt.

B.3.1 Loading data memory, program memory, queues, etc.

After the simulator configures the basic memory structure, it prompts (with "SIM) ") for a command

```
SIM) lcode (cr)
_file  scorv0.obj
```

The command reads the object file (.obj) and the Symbol table file (.sym) created by the meta-assembler and loads it in the present command cell's program memory..

To load the data memory of the command cell by integer numbers, type

```
SIM) ldm(cr)
_Address 0(cr)
_Length 5(cr)
10000 0abcdfh 01010101b 1234 33333(cr)
```

Thus hex, octal, binary or decimal input can be given.

To load the data memory with floating-point data, type

```
SIM) ldm -f(cr)
 _Address:0(cr)
_Length: 5(cr)
1.23 1.2e12 1234567 1.00e24 24.23456
```

B.3.2 Command files

When starting a simulator session user may wish to execute the same set of commands to bring the simulator to a known state. Instead of entering these command strings every time, user can create a command file which contains these command strings. The command strings appear in the command file exactly as if typed interactively.

To execute commands in a command file, type

```
SIM) cmdfile(cr)
_File conv.cmd(cr)
```

An example of the command file is given below

```
element
0
lcode
sconv0.obj
lx -f
0
2
1.3
3.4
element
1
lcode
sconv1.obj
ldm -f -lf
0
4
conv.dat
```

Note that, while giving data inputs for load commands each data should be given on the separate line.

B.3.3 Hardware configuration

INTERRUPTS

The simulator has internal cycle counters for each cell and IFU , which can be set to model interrupting devices. User can activate any one of the 8 interrupt sources for each cell and IFU by specifying an interval time between two interrupts. When the interval time expires, an interrupt is issued to corresponding cell's sequencer. On Simulator startup , all interrupt sources are disabled. When enabling an interrupt user should specify the interrupt level (1 to 8) and the interval time period.

```
SIM) setint (cr)
_number  1 (cr)
_period (cycles)  20 (cr)
```

To disable an interrupt source, simply set the interrupt period to 0.

RESET:

Reset command resets the system to starting position. All the counters, and program counters of all the cells and IFU are reset to 0.

B.3.4 OPERATION MODES

After configuring the hardware, the simulator is ready to run a user program. The simulator operates in one of the three modes: Emulator mode, Extend mode, and Single Step mode.

The default and starting operation mode is the emulator mode. In this mode, the simulator runs at full speed and halts only when a break condition is encountered. Break conditions include break points, simulator errors and the user

Start the simulator by typing

SIM) run<cr>

When the simulator hits a break condition and halts, the simulator displays a message to inform you what caused the halt

B.3.5 DISPLAY MODE

The simulator can display

-Register file

-Data memory

-Program memory

-Addr, X , Y Queues

-Scratch pad memory

-ALU , Multiplier registers

for the command cell by giving commands regf, dm, pm , addrq, xq, yq, spad, areg, mreg respectively. For example , to display register file contents in floating point format, type

SIM) regf -f<cr>

_Address 00<cr>

10 locations from address 0 will be displayed in the floating point format.

To display

-X memory

-Ya memory

-Yb1 memory

-Yb2 memory

the command cell need not be the cell 0 (IFU). The commands to display these memories are x, ya, yb1, yb2 respectively.

The Yb memory can be read sequentially , as it was loaded before, by the command rdyb

B.3.6 MODIFY COMMANDS

To modify program counter for the command cell type,

```
SIM) setpc(cr)
_address  12h(cr)
```

To change the command cell, type

```
SIM) element(cr)
_Current command cell  0
_Number  1(cr)
```

To change/display cycle count, type

```
SIM) cycle(cr)
_Current cycle count  0
_count  12(cr)
```

B.3.7 Ending a Simulator Session

At the end of a simulation session, user can save a segment of data memory or ya or yb memory content by dumping it into a file

```
SIM) dumpdm [option](cr)
_Address  0(cr)
_Length  10(cr)
_file  dump.dat(cr)
```

The option is '-f' ,to dump a floating point data. The default is integer data.

To terminate a simulation session, type

The Yb memory can be read sequentially , as it was loaded before, by the command rdyb

B.3.6 MODIFY COMMANDS :

To modify program counter for the command cell type,

```
SIM) setpc(cr)
_address  12h(cr)
```

To change the command cell, type

```
SIM) element(cr)
_Current command cell  0
_Number  1(cr)
```

To change/display cycle count, type

```
SIM) cycle(cr)
_Current cycle count  0
_count  12(cr)
```

B.3.7 Ending a Simulator Session :

At the end of a simulation session, user can save a segment of data memory or ya or yb memory content by dumping it into a file :

```
SIM) dumpdm [option](cr)
_Address  0(cr)
_Length  10(cr)
_file  dump.dat(cr)
```

The option is '-f' ,to dump a floating point data. The default is integer data.

To terminate a simulation session, type

SIM) exit(cr)

_Verify Y(cr)

and you return to the host system

B.3.8 HELP :

To recall what commands are supported by the simulator, user can list all supported commands with the help command.

SIM) help(cr)

To get a help on a particular command type

SIM) help commandname(cr)

B.4 *Simulator Command summary*

This session lists all simulator commands.

DISPLAY CONTROL COMMANDS

fo -Forces previous display to scroll forward.

ba -Forces previous display to scroll back.

ya -Displays Ya memory

dm -Displays data memory

x -Displays x memory

xq -Displays x queue

yq -Displays y queue

addrq -Displays address queue.

regf -Displays register file

pm -Displays Program memory

seqreg -Displays sequencer registers.

yb1 -Displays Yb1 memory.

yb2 -Displays Yb2 memory.

agreg -Displays address register.

mreg -Displays Multiplier registers.

areg -Displays ALU registers.

rdyb -Reads Yb memory from the present counter position.

dispint -Displays the interrupt sources.

dispb -Displays break points.

EXIT COMMAND

exit -Exits from the Simulator.

FILE CONTROL COMMANDS AND LOAD COMMANDS

lx -Loads X memory.(Data can be loaded from a file by giving -f1 switch)

lya -Loads Ya memory.(Data can be loaded from a file by giving -f1 switch)

lyb -Loads Yb memory.(Data can be loaded from a file by giving -f1 switch)

ldm -Loads data memory.(Data can be loaded from a file by giving -f1 switch)

lcode -Loads program/data memory from obj file.

dumpdm -Forces DM memory segment dump to a file.

dumpya -Forces Ya memory segment dump to a file.

dumpyb1 -Forces Yb1 memory segment dump to a file.

dumpyb2 -Forces Yb2 memory segment dump to a file.

cmdfile -Executes simulator commands found in a command file.

BREAK CONTROL COMMANDS

setb -Sets a PM break address.

clrb -Clears one/all break points.

OPERATION CONTROL COMMANDS

extend -Invokes extend mode.

emulate -Invokes emulate mode.

ss -Invokes single step mode.

MODIFY COMMANDS

clryb -Clears Yb memory read/write counters.

element -Displays/changes command cell number.

setpc -Sets the PC.

cycle -Displays/changes the cycle count.

CONFIGURATION CONTROL COMMANDS

reset -Simulates hardware system reset.

setint -Activates the interrupt source.

EXECUTION CONTROL COMMANDS

run -Starts user program running.

. <cr>

HELP COMMAND

help -Displays command list or help for a command.

APPENDIX C

```
; THE DEFINITION FILE FOR THE INTERFACE UNIT (IFU).

TITLE INTERFACEUNIT
DEFN_FILE "SIFUF.DAT"
WORD WIDTH 56 BITS
DATA_LOC_WIDTH 32
PROGRAM_LOC_WIDTH 56
;1
#sequencer:    FIELD 49-55,WIDTH 7 DEFAULT cont
  OPCODE_BIT_NOTATIONS [
      kk (       ;Conditins
            00 unconditional
            01 notflag
            10 flag
            11 sign
      )
      cc (     ;Selects the relevent register(R3-R0)and
            C0-C3      ;/or counter(C3-C0).
      )
      ii (;Decides number to be added incase of
            I0-I3       ; AIRSP instruction.
      )
      ]
  INSTR_NOT_AVAILABLE [
            jrc(sign)(c0)
            jrc(sign)(c1)
            jrc(sign)(c2)
            jrc(sign)(c3)
            branch(sign)(c0)
            branch(sign)(c1)
            branch(sign)(c2)
            branch(sign)(c3)
      ]
   BRANCH_INSTR_WHICH_NEED_DATA [
      ABSOLUTE (
            jsa
            jda
            jdrst
            )
      RELATIVE (
            jsr
            jdr
            )
      ]
   VALUES [
;jump & branch instruction

      jncnf       0010101
```

```
      irmbc       0010011
      irmbs       0010010
      disir       0010110
      enair       0110110
      slir        0010111
      stir        0110111
      slrivp      0011101

 ;relative address width controls:
      rel16       0100100
      rel12       0100111
      rel8        0100110

 ;miscellaneous instructions:
      cont        0000000
      idle        0010000
      ihc         0100101
      wcs         0100000
      ]
;2
#data: FIELD 32-47,WIDTH 16,DEFAULT zero

  VALUES [
      zero        0000000000000000
]
;3
#address_generator: FIELD 22-31,WIDTH 10,DEFAULT nop

  OPCODE_BIT_NOTATIONS [
        cc ( ;Comparison Register number
            c0-c3
            )
        rrr ( ;Three bit Address register number
            R0-R7
            )
        rrrr ( ;Four bit Address register number
            r0-r15
            )
        bb ( ;Base (offset) register number
            b0-b3
            )
        ii ( ;Initialisation register number
            i0-i3
            )
        pp ( ;Two bit precision code
            p0-p3
            )
        x ( ;One bit control bit
            x0-x1
            )
      ]
 VALUES [
  ;looping instructions
      yinc        1011ccrrrr
```

```
      ydec        1010ccrrrr
      yadd        11ccbb1rrr
      ysub        11ccbb0rrr
  ;register transfer instructions:
      yrtr        000101rrrr
      yrtb        0011bbrrrr
      yrtc        0010ccrrrr
      dti         0000111111
      itr         1000iirrrr
      btr         0100bbrrrr
      rtd         000100rrrr
      ctd         00001100cc
      btd         00001101bb
      itd         00001110ii
  ;logical and shift instructions
      yor         0111bbrrrr
      yand        0110bbrrrr
      yxor        0101bbrrrr
      yasr        000111rrrr
      ylsl        000110rrrr
      rst         0000000001
      dtcr        0000101110
      crtd        0000101111
      seti        0000100iix
      setp        00001010pp
      sety        000001001x
      selr        000001101x
      selb        000001100x
      setu        000001011x
      seta        000001010x
      wra         0000101100
      rda         0000101101
      lda         0000011110
      ydty        0000011111
      yrev        1001bbrrrr
      nop         0000000000
      ]
;4
#ag_inpdata_select: FIELD 21-21,WIDTH 1,DEFAULT nodsel

  VALUES [
       nodsel   0
      dsel      1
      ]
;5
#ag_air_select: FIELD 20-20,width 1,default noair
VALUES [
      air       1
      noair     0
      ]
;6
#write_yb_mem: FIELD 19-19 width 1, default nowryb
VALUES [
      nowryb    0
```

```
      wryb        1
      ]
;7
#write_x_queue: FIELD 18-18 width 1 default nowrxq
VALUES [
      nowrxq      0
      wrxq        1
]
;8
#read_yb_mem: FIELD 17-17 width 1 default nordyb
VALUES [
      nordyb      0
      rdyb        1
]
;9
#read_ya_mem: FIELD 16-16 width 1 default nordya
VALUES [
      nordya      0
      rdya        1
      ]
;10
#read_x_mem: FIELD 15-15 width 1 default nordx
VALUES [
      nordx       0
      rdx         1
      ]
;11
#clear_x_mem: FIELD 14-14 width 1 default noclrx
VALUES [
      noclrx      1
      clrx        0
      ]
;12
#clear_ya_mem: FIELD 13-13 width 1 default noclrya
VALUES [
      noclrya     0
      clrya       1
      ]
;13
#clear_yb_mem: FIELD 12-12 width 1 default noclryb
VALUES [
      noclryb     0
      clryb       1
      ]

;14
#data_enable: FIELD 11-11 width 1 default noken
VALUES [
      noken       0
      ken         1
      ]
;15
#flag_mux: FIELD 8-10 width 3 default noflag
VALUES [
```

```
      noflag      000
      sync        001
      cmpz        010
      end_op      011
      flg         100
      abc5        101
      abc6        110
      abc7        111
      ]
;16
#data_latch_enable FIELD 7-7 width 1 default noden
VALUES [
      den         1
      noden       0
      ]
;17
#data_strobe field 6-6 width 1 default nodstb
VALUES [
      dstb        1
      nodstb      0
      ]
;18

#write_y_queue FIELD 4-4 width 1 default nowryq
VALUES [
      nowryq      0
      wryq        1
      ]
;19
#write_address_queue FIELD 3-3 width 1 default nowraddrsq
VALUES [
     nowraddrsq 0
      wraddrsq   1
      ]
;20
#y_bus_direction FIELD 2-2 width 1 default celli_n
      VALUES [
      celli_n     1
      celln_1     0
      ]
;21
#write_x_mem FIELD 1-1 width 1 default nowrx
VALUES [
      nowrx       0
      wrx         1
      ]
;22
#read_data_mem FIELD 0-0 width 1 default norddm
VALUES [
      norddm      0
      rddm        1
      ]
```

```
; THE DEFINITION FILE  FOR THE CELL UNIT.

TITLE CELL
DEFN_FILE "SCELLF DAT"
WORD WIDTH 136 BITS
DATA_LOC_WIDTH 32
PROGRAM_LOC_WIDTH 136
;1
#sequencer     FIELD 129-135,WIDTH 7 DEFAULT cont
  OPCODE_BIT_NOTATIONS [
       kk (  ,Conditins
           00 unconditional
           01 notflag
           10 flag
           11 sign
      )
         cc (  ,Selects the relevent register(R3-R0)and
           C0-C3  ,/or counter(C3-C0)
      )
         ii (  ,Decides number to be added incase of
           I0-I3  , AIRSP instruction
      )
      ]
   INSTR_NOT_AVAILABLE [
      jrc(sign)Xc0)
      jrc(sign)Xc1)
      jrc(sign)Xc2)
      jrc(sign)Xc3)
      branch(sign)Xc0)
      branch(sign)Xc1)
      branch(sign)Xc2)
      branch(sign)Xc3)
      ]
   BRANCH_INSTR_WHICH_NEED_DATA [
      ABSOLUTE (
           jsa
           jda
           jdrst
           )
      RELATIVE (
           jsr
           jdr
           )
      ]
   VALUES [
;Jump & branch instruction

      jpcof       0010101
      jpcnf       0110101
      jtwo        101kk01
      jda         111kk11
      jdr         111kk01
      jrc         110kkcc
```

```
        jdi         101kk10
        jdrst       10011cc
        jrs         11011cc
        jsa         111kk00
        jsr         111kk10
        rtn         101kk11
        branch      100kkcc

  ;stack operation
 ;subroutine stack

        psdss       0011110
        ppssd       0111110
        wrssp       0001110
        rdssp       0101100
        dssp        0000010

 ;register stack
        sgsp        0000111
        slsp        0000110
        rdrsp       0101111
        wrrsp       0001100
        pspc        0100011
        psgsp       0000101
        ppgsp       0000100
        psdrs       0011111
        pprsd       0111111
        airsp       01010₁₁
        sirsp       0001111
        s4rsp       0111100

 ;status register operations
       rdsr         0101110
       wrsr         0011100
       pssr         0100001
       ppsr         0100010

 ;counter operations
       wrcntr       01110cc
       clrs         0010100
       sets         0110100
       pscntr       00010cc
       ppcntr       00110cc
       docntr       01100cc
       ifcdec       101kk00
```

```
    disir       0010110
    enair       0110110
    slir        0010111
    stir        0110111
    slrivp      0011101

;relative address width controls
    rel16       0100100
    rel12       0100111
    rel8        0100110

;miscellaneous instructions
    cont        0000000
    idle        0010000
    ihc         0100101
    wcs         0100000
    ]
;2
#data: FIELD 113-128,WIDTH 16,DEFAULT zero

 VALUES [
    zero        0000000000000000
    ]

;3
#data_enable  FIELD 112-112 width 1 default noken
VALUES [
    noken       0
    ken         1
    ]
;4
#data_latch_enable  FIELD 111-111 width 1 default noden
VALUES [
    den         1
    noden       0
    ]
;5
#data_strobe  field 110-110 width 1 default nodstb
VALUES [
    dstb        1
    nodstb      0
    ]
;6
#address_generator: FIELD 100-109,WIDTH 10,DEFAULT nop

 OPCODE_BIT_NOTATIONS [
      cc (  ;Comparison Register number
          c0-c3
    )
      rrr (  ;Three bit Address register number
          R0-R7
    )
      rrrr (  ;Four bit Address register number
          r0-r15
```

```
    )
      bb ( ,Base (offset) register number
          b0-b3
    )
      ii ( ,Initialisation register number
          i0-i3
    )
      pp ( ,Two bit precision code
          p0-p3
    )
      x ( ,One bit control bit
          x0-x1
    )
    ]
VALUES [
;looping instructions
  yinc          1011ccrrrr
  ydec          1010ccrrrr
  yadd          11ccbb1rrr
  ysub          11ccbb0rrr
    ;register transfer instructions
  yrtr          000101rrrr
  yrtb          0011bbrrrr
  yrtc          0010ccrrrr
  dti           0000111111
  itr           1000iirrrr
  btr           0100bbrrrr
  rtd           000100rrrr
  ctd           00001100cc
  btd           00001101bb
  itd           00001110ii
    ;logical and shift instructions
  yor           0111bbrrrr
  yand          0110bbrrrr
  yxor          0101bbrrrr
  yasr          000111rrrr
  ylsl          000110rrrr

  rst           0000000001
  dtor          0000101110
  crtd          0000101111
  seti          000010011x
  setp          00001010pp
  sety          000001001x
  selr          000001101x
  selb          000001100x
  setu          000001011x
  seta          000001010x
  wra           0000101100
  rda           0000101101
  lda           0000011110

  ydly          0000011111
  yrev          1001bbrrrr
```

```
   nop            0000000000
   ]
;7
#ag_inpdata_select  FIELD 99-99,WIDTH 1,DEFAULT nodsel

 VALUES [
    nodsel      0
   dsel         1
   ]
;8
#ag_air_select: FIELD 98-98,width 1,default noair
VALUES [
   air          1
   noair        0
   ]

;9
#flag_mux: FIELD 95-97 width 3 default flg
VALUES [
   flg          000
   ompz         001
   lessthan     010
   equal        011
   greaterthan  100
   invalop      101
   ovrflo       110
   underflo     111
   ]
;10
#write_scratch_pad_mem: FIELD 94-94 width 1, default nowrspmem
VALUES [
   nowrspmem    0
   wrspmem      1
   ]
;11
#read_scratch_pad_mem : FIELD 93-93 width 1 default nordspmem
VALUES [
   nordspmem    0
   rdspmem      1
   ]
;12
#write_x_queue: FIELD 92-92 width 1 default nowrxq
VALUES [
   nowrxq       0
   wrxq         1
   ]
;13
#write_y_queue: FIELD 91-91 width 1 default nowryq
VALUES [
   nowryq       0
   wryq         1
   ]
;14
#write_address_queue: FIELD 90-90 width 1 default nowraddrq
```

```
   retransy      1
   ]
;22
#read_address_queue  FIELD 81-81 width 1 default nordaddrq
VALUES [
   nordaddrq     0
   rdaddrq       1
   ]
;23
#clear_address_queue  FIELD 80-80 width 1 default noclraddrq;Master reset.
; Read/Write counters are reseted
VALUES [
   noclraddrq    0
   clraddrq      1
   ]
;24
#retransmit_address_queue  FIELD 79-79 width 1 default noretransaddrq
;Retransmt
; Only Read counter is reseted
VALUES [
noretransaddrq  1
retransaddrq    0
]
;25
#address_crossbar_dmaddress  FIELD 78-78 width 1 default axbdm_agi
VALUES [
   axbdm_agi     0
axbdm_addrqi     1
]
;26
#address_crossbar_spaddress  FIELD 77-77 width 1 default axbsp_agi
VALUES [
   axbsp_agi     0
   axbsp_addrqi  1
]
;27
#y_queue_input_select  FIELD 76-76 width 1 default yprevious
VALUES [
   ycurrent      1
   yprevious     0
]
;28
#mul_port_selection  FIELD 74-75 width 2 default noinput
VALUES [
   noinput       00
   mul_aen       01
   mul_ben       10
   ]
;29
#alu_output_enable:  FIELD 73-73 width 1 default noaoen
VALUES [
   noaoen        0
   aoen          1
   ]
```

```
;30
#mul_output_enable FIELD 72-72 width 1 default nomoen
VALUES [
   nomoen       0
   moen         1
   ]
;31
#mul_single_precision_mode FIELD 71-71 width 1 default msp
   VALUES [
   msp          1
   mfixed       0
   ]
;32
#reg_file_write_a FIELD 70-70 width 1 default reg_awrinh
;Write register port A
values [
   reg_awrinh   0
   reg_awr      1
   ]
;33
#reg_file_write_b FIELD 69-69 width 1 default reg_bwrinh
;Write register port B
values [
   reg_bwrinh   0
   reg_bwr      1
   ]
;34
#reg_file_write_e FIELD 68-68 width 1 default reg_ewrinh
;Write register port E
values [
   reg_ewrinh   0
   reg_ewr      1
   ]
;35
#reg_file_read_c FIELD 67-67 width 1 default reg_ctri
;Read register port C
values [
   reg_ctri     0
   reg_crd      1
]
;36
#reg_file_read_d FIELD 66-66 width 1 default reg_dtri
;Read register port D
values [
   reg_dtri     0
   reg_drd      1
   ]
;37
#reg_file_read_e FIELD 65-65 width 1 default reg_etri
;Read register port E
values [
   reg_etri     0
   reg_erd      1
]
```

```
;38
#reg_file_address_a FIELD 58-64 width 7 default zeroadd_a
 OPCODE_BIT_NOTATIONS [
  ######## (
              #0-#127
  )
  ]
values [
  zeroadd_a   0000000
  a_addr      ########
  ]
;39
#reg_file_address_b FIELD 51-57 width 7 default zeroadd_b
 OPCODE_BIT_NOTATIONS [
  ######## (
              #0-#127
              )
  ]
values [
  zeroadd_b   0000000
  b_addr      ########
  ]
;40
#reg_file_address_c FIELD 44-50 width 7 default zeroadd_c
 OPCODE_BIT_NOTATIONS [
  ######## (
              #0-#127
              )
  ]
values [
  zeroadd_c   0000000
  c_addr      ########
  ]
;41
#reg_file_address_d FIELD 37-43 width 7 default zeroadd_d
 OPCODE_BIT_NOTATIONS [
  ######## (
              #0-#127
              )
  ]
values [
  zeroadd_d   0000000
  d_addr      ########
  ]
;42
#reg_file_address_e FIELD 30-36 width 7 default zeroadd_e
 OPCODE_BIT_NOTATIONS [
  ######## (
              #0-#127
              )
  ]
values [
  zeroadd_e   0000000
  e_addr      ########
```

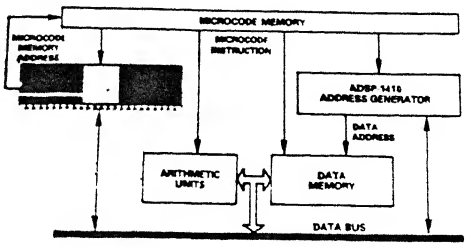
```
;48
#write_ya_mem FIELD 14-14 width 1 default nowrya
VALUES {
   nowrya       0
   wrya         1
   }
;49
#aluFIELD 5-13 width 9 default anop
VALUES {
   anop         000000000
   iadd         001000011
   isubb        001001011
   isuba        001000111
   aandb        000010010
   aorb         000100010
   axorb        000110010
   sadd         111000011
   ssubb        111000111
   ssuba        111001011
   scomp        111001111
   }
;50
#alu_sp_buffer_direction FIELD 4-4 width 1 default aout_to_in
VALUES {
   aout_to_in   1
   ain_to_out   0
   }
;51
#alu_sp_buffer_enable FIELD 3-3 width 1 default aluout_bufdisable
VALUES {
   aluout_bufdisable    0
   aluout_bufen 1
   }
;52
#alu_aport_selection FIELD 2-2 width 1 default noainput
VALUES {
   noainput     0
   alu_aen      1
   }
;53
#alu_bport_selection FIELD 1-1 width 1 default nobinput
VALUES {
   nobinput     0
   alu_ben      1
   }
;54
#x_queue_input_select  FIELD 0-0 width 1 default xprevious
VALUES {
   xcurrent     1
   xprevious    0
   }
```

ANALOG DEVICES

Word-Slice Program Sequencer

ADSP-1401

FEATURES
16-Bit Microcode Addressing Capability
Look-Ahead™ Pipeline
Extensive Interrupt Processing, With Ten On-Chip Interrupt Vectors
70ns Cycle Time; 25ns Clock-to-Address Delay
64-Word RAM for Storing:
  Subroutine Linkage
  Jump Addresses
  Counters
  Status Register
375mW Maximum Power Dissipation with CMOS Technology
48-Pin Ceramic or Plastic DIP and 52-Lead Plastic Leaded Chip Carrier



WORD-SLICE™ MICROCODED SYSTEM WITH ADSP-1401

GENERAL DESCRIPTION

The ADSP-1401 is a high-speed microprogram controller optimized for the demanding sequencing tasks found in digital signal processors and general purpose computers. In addition to high speed (25ns clock-to-address delay) and large addressing range (64K of program memory), this Word-Slice™ component has unique features that make it highly versatile.

- on-chip storage and control of ten prioritized and maskable interrupts
- four decrementing event counters
- absolute, relative and indirect addressing capability
- download capability (writeable control store) and
- a dynamically configurable 64-word RAM.

The ADSP-1401 microprogram sequencer's main task is to provide the appropriate microprogram addressing to support programming requirements (e.g., looping, jumping, branching, subroutines, condition testing and interrupts). An internal Look-Ahead pipeline, controlled by both phases of the clock, allows the ADSP-1401 to satisfy these requirements at very high speed.

During each micro-instruction, the ADSP-1401 monitors the conditions and instructions to determine the next microprogram address. This address can come from one of several sources: the stack, the jump address space in the RAM, the data port, the interrupt vectors, or the microprogram counter. An extensive set of conditional instructions are also available, including jumps, branches, subroutines, interrupts, and writeable control store.

The ADSP-1401's internal 64-word RAM is user-configurable into three regions; subroutine stack, register stack and indirect jump address space. The subroutine stack is used for linking interrupts and subroutines and, during their execution, allow storage of system states. The register stack allows association of unique jump addresses with various levels of interrupts and subroutines (both local and global stacks are provided). Indirect jump capability is also supported, addressing for which is provided at the data port.

Interrupts are handled entirely on chip. The ADSP-1401's internal interrupt control logic includes registers for eight external (user) interrupt vectors, a mask register, and a priority decoder. Two additional vectors are reserved for internally-generated interrupts resulting from counter underflow and stack limit violation. A stack limit violation is caused by stack overflow, underflow or collision. A mechanism is provided for recovering from stack violations.

The ADSP-1401's four decrementing 16-bit counters are used to track loops and events. These counters generate a signal when negative. This negative condition is used by several conditional instructions and can also trigger an internal interrupt.

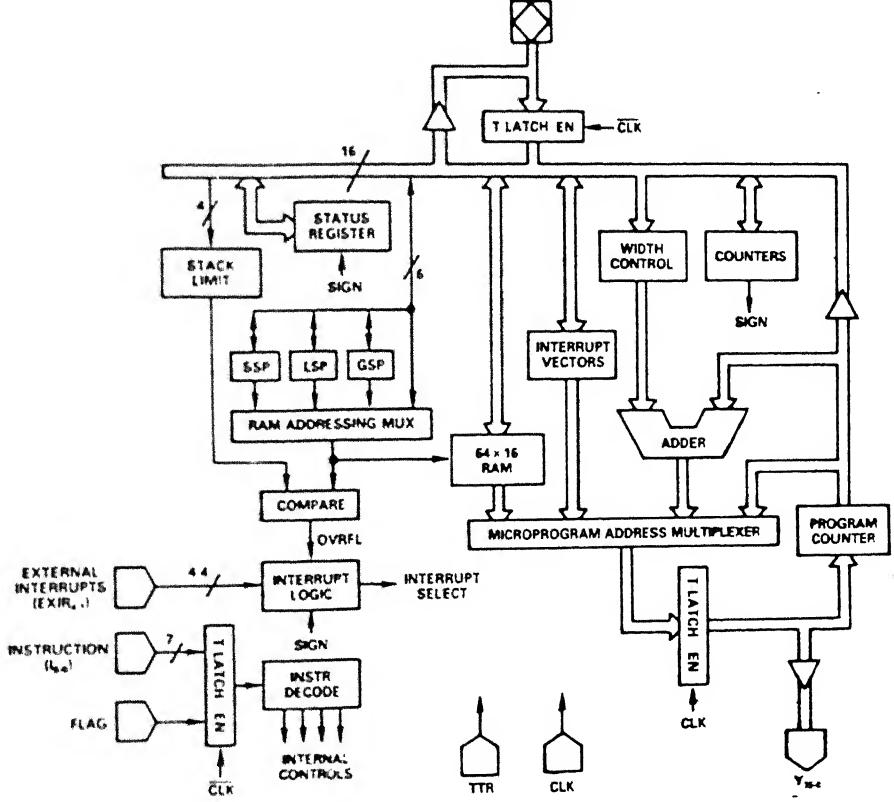


Figure 1 ADSP-1401 Block Diagram

ADDRESSING MODES

Direct: both absolute and relative
Indirect: from internal RAM

HARDWARE FEATURES

Instruction Port
Bidirectional Data Port
Four Input Address Multiplexer
Three Stack Pointers
Four Event Counters
Condition Flag
Eight Prioritized and Maskable User Interrupts
TTR Pin:
- Trap
- Three-State
- Reset

INSTRUCTION TYPES

Jumps and Branches
Stack Operations
Status Register Operations
Counter Operations
Interrupt Control
Relative Address Width Controls
Instruction Hold Control
Writeable Control Store
Dedicated Counter Underflow Interrupt
Dedicated Stack Overflow Interrupt

ADSP-1401 PIN ASSIGNMENTS

Pin Name	Description
I_6-I_0	The 7-bit microinstruction controlling the ADSP-1401.
$Y_{15}-Y_0$	Output bus which provides addresses to the microprogram memory.
$D_{15}-D_0$	Bidirectional Data bus for transferring data to or from the ADSP-1401.
$EXIR_{4-1}$	Four external interrupt request lines. Note that internal circuitry supports 8 interrupts with the aid of an external 2 to 1 multiplexer.
CLK	External clock input
FLAG	An input used for conditional instructions. Its source is usually a condition multiplexer.
TTR	A multi-purpose pin accommodating traps, output disable and reset.
V_{DD}	+5 Volt supply.
GND	Ground.

1.0 ARCHITECTURE

1.1 Look-Ahead Pipeline

Logically, the Look-Ahead pipeline is split into two halves: the first, located at the instruction and data ports; and the second, located at the address port. Each half of the pipeline (input vs. output) has a transparent latch which operates out of phase with the other; the address latch is transparent during the first half of the cycle (clock HI), while the input latches (instruction and data) are transparent during the second half of the cycle (clock LO). This complementary arrangement allows new instructions to be decoded (in preparation for the following cycle) while the program address for the current cycle is held steady.

1.2 Instruction Port

The instruction port receives 7-bit instructions defining the next operation to perform from microcode. The ADSP-1401 has a built-in Look-Ahead pipeline latch, eliminating the need for an external microcode latch to hold instructions. This implementation has the further benefit of allowing instruction "look-ahead"; the sequencer is able to decode the next instruction during execution of the current cycle. During the "look-ahead" period, the sequencer precalculates the next address, allowing its output as early as possible in the next cycle.

External instructions are internally latched during clock HI, and passed directly to the instruction decoder during clock LO (transparent phase), thus, implementing the first half of the Look-Ahead pipeline latch.

The use of the instruction hold mode (see: Instruction Set Description, 2.7; and Instruction Hold Control, appendix 4.1) allows an instruction to be held in the instruction latch for execution over several cycles (freeing microcode for use by other devices).

1.3 Address Port and Multiplexer Sources

The address port provides 16-bit program addresses with three-state drivers designed for driving large microcode memories. Addresses come from a four-to-one microprogram address multiplexer. Between the multiplexer and output port is a transparent latch which is transparent during clock HI and latched during clock LO, permitting addresses to be output as early as possible during phase one (clock HI) while holding the address constant during phase two (clock LO) – implementing the second half of the Look-Ahead pipeline latch.

Inputs to the microprogram address multiplexer are the:

- 16-Bit Program Counter
- 16-Bit Adder
- Interrupt Vector File and
- Internal 64-Word RAM.

Addressing Modes

The ADSP-1401 supports two addressing modes: direct and indirect. The direct addressing mode uses the internal adder to generate either absolute addresses from the data port (without modification) or relative addresses from the program counter (with or without extension: see Status Register, 1.4.4). The indirect addressing mode uses the lower order bits at the data port to access the contents of internal RAM for output.

Output Drivers

The address port output drivers are always active unless placed in the high-impedance state by the IDLE instruction or appropriately asserting the TTR pin (see TTR Pin, 1.7). This allows other devices to supply microcode addresses, which is particularly useful in multi-tasking or context switching applications where several ADSP-1401s may be sharing common microcode memory.

1.3.1 Program Counter

The program counter (PC) consists of a 16-bit incrementing counter. For most instructions, the PC is incremented by the end of the cycle (post-increment) as follows:

$$PC <= \text{output address} + 1.$$

1.3.2 Adder and Width Control

For absolute jumps, data from the data port is passed unchanged through the adder directly to the microprogram address port. For relative jumps, a twos complement offset is supplied from the data port and added with the 16-bit PC. Since the PC normally points to the next instruction, the jump distance is (offset + 1) from the jump instruction. See Status Register (1.4.4) for more details.

The width control block permits microcode width to be reduced in systems not requiring full, 16-bit jump distances. Internal width control logic sign-extends reduced offsets of 8- and 12-bits to full 16-bit precision, accommodating jumps in either direction (positive or negative displacement).

1.3.3 Interrupt Vector File

Ten prioritized interrupt vectors may be stored in the interrupt vector file. The associated interrupts are internally latched and may be individually masked or entirely disabled by the "Disable Interrupts" (DISIR) instruction. The highest priority interrupt vector displaces the usual address on the next cycle following its detection. See Interrupts (1.4.3) for more details.

1.3.4 Internal RAM

Any of the 64 words of RAM may be output on the address port. Four distinct address sources may access the RAM.

- Local Stack Pointer
- Global Stack Pointer
- Subroutine Stack Pointer and
- Lower Order Data Port Bits.

The use of internal RAM and its various address sources are described in section 1.4.2.

1.4 Bidirectional Data Port

The 16-bit bidirectional data port (D_{15-0}) supplies direct or indirect jump addresses and permits loading or dumping of all internal registers. The input data latch freezes incoming data (for counter or register writes executed during that cycle) during the first half-cycle (clock HI) and is transparent for the remainder of the cycle. The output data driver asserts output data *only* during the first half-cycle of a data output instruction and is independent of the address port drivers. This complementary I/O arrangement permits data to be output from the sequencer (as in a read register instruction) during the first half-cycle while accommodating external data setups (for the next cycle) during the second half-cycle.

1.4.1 Counters

Four independent 16-bit counters are provided for maintaining loops and event tracking. These counters hold twos complement values that may be decremented or preloaded through dedicated instructions. The sign bit associated with the most recently used counter, prior to its decrement, is always saved in the status register (SR). Simultaneously, the sign bit is also made available to control various conditional instructions or for asserting the lowest priority interrupt, IR_0, reserved for counter underflow (see Instruction Set Description, 2.0, and Interrupts, 1.4.3).

Note that interrupt IR_0 is primarily used for ending writeable control store downloads (see Instruction Set Description – WCS, 2.7). Use of IR_0 in the context of a "Decrement Counter and Interrupt on Underflow" operation represents the worst case instruction and flag setup times because of the additional overhead in processing the interrupt after determining whether the counter was underflowed. These setup times are specified two ways:

1. all conditions used
2. IR_0 masked

The source of SIGN applied to the condition test depends upon the type of instruction used (see Instruction Set Description, 2.1). Two possibilities exist:

1. If an *explicit* counter is selected, then the sign applied is that of the counter, prior to the decrement.
2. If *no counter* is selected, then the sign applied is implicitly that of the *status register*, SR.

1.4.2 Internal RAM

The ADSP-1401's internal 64-word RAM implements two distinct stacks: a Subroutine Stack (SS) and a Register Stack (RS). The subroutine stack has a dedicated Subroutine Stack Pointer (SSP), while the register stack shares two pointers: the Local Stack Pointer (LSP) and the Global Stack Pointer (GSP). The three stack pointers are each held in 6-bit, preloadable, up/down counters.

Upon reset, TTR pin held HI for three cycles, see TTR Pin, 1.7, the SSP is initialized to 0, top of RAM. The RS pointers (LSP and GSP) are typically configured as shown in Figure 2 using the "Write RSP" instruction (WRRSP). The SSP pushes down while the RS pointers push up. Selection of the active RS pointer (LSP or GSP) is made in the status register.

Stack overflow detection is provided via a stack limit register to protect software integrity and allow stack expansion (see Instruction Set Description – SLRIVP, 2.5).

Each RS pointer may be explicitly initialized by performing the "Write RS Pointer" (WRRSP) instruction. The LSP should be located above the GSP, allowing the local stack to grow upwards as the level of nested subroutines increases. Finally, indirect jump address space (as needed) should be reserved below the global stack.

The sequencer will generate a stack underflow interrupt whenever RAM location zero is popped. This facility may be used in support of stack paging. IV_0 should be masked if not using stack paging, allowing location zero to be used as the first stack location without interrupting. When using paged stacking, location zero must be reserved as an underflow buffer to avoid a subsequent

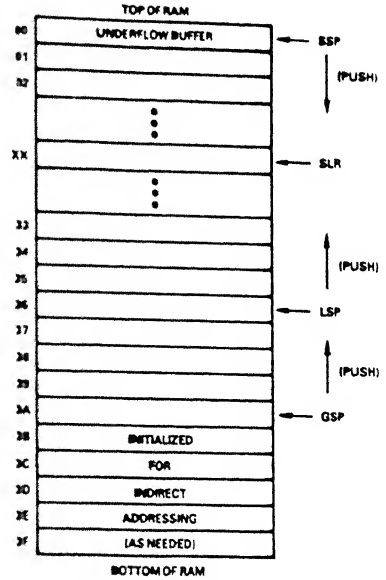


Figure 2. Typical RAM Initialization

Register Stack Pointers (LSP and GSP)

Upon entering a routine, up to four jump addresses may be pushed onto the register stack. A Push onto the register stack first decrements the RS pointer (either LSP or GSP, depending upon the status register) and then writes the appropriate data to RAM. A Pop from the register stack first reads the RAM location and then increments the RS pointer (LSP or GSP).

Four registers are available within context of any routine which are addressed relative to the stack pointer (LSP or GSP) by the two LSBs of the relevant instruction. For example, the instruction:

IF CONDITION, JMP R_2

accesses the location (LSP+2 or GSP+2) in RAM as the conditional address source. Prior to exiting a routine, local or global registers can be effectively removed from the RS by the "ADD i TO RSP" (AIRSP) instruction (see Instruction Set Description, 2.2).

Often, the same set of jump addresses are used by several different routines. The GSP is available for addressing these common registers — conserving RAM space and eliminating repeated stack pushes and pops. Global registers can be pushed, popped, and used by conditional instructions in the same way that local registers are handled. In addition, the GSP can itself be pushed and popped to/from the subroutine stack, allowing different routines to access different subsets of the global stack area.

Subroutine Stack Pointer (SSP)

A Push onto the SS (jump subroutine or interrupt) first increments the SSP and then writes the return address to RAM. A pop from the SS first reads the return location and then decrements the SSP, effectively removing the data from the stack (although the data remains in RAM). For interrupts, the return address is the one that would have been output in the cycle when the

interrupt vector was output. For subroutine jumps, the return address is the instruction immediately following the subroutine call. For further information, see Return from Interrupt with Pending Interrupt, appendix 4.2, and the Instruction Set Description, 2.0.

The subroutine stack can also be used to save key program parameters such as the status register, GSP, or counter values. After entering a new routine, critical parameters from the calling routine are pushed onto the stack, thus freeing the associated hardware for use by the new routine. Prior to the end of the routine, the original parameters are restored with their former values for continued use by the calling routine.

The Stack Usage Example (appendix 4.3) illustrates the state of RAM after three subroutine calls.

Stack Limit Register and Stack Overflow.
The preloadable Stack Limit Register (SLR) and associated circuitry warns the user of impending stack overflows, permitting stack overflow recovery. The highest priority interrupt, IR_9, is assigned to stack overflow, although it may be masked. A stack overflow interrupt will occur under any of the following three circumstances:

- a push causing the SSP to increment to the value in the stack limit register
- a pop from SS location 00 (underflow)
- a push causing the RS pointer (USP or GSP) to decrement to the value in the stack limit register + 3

The three location buffer between the SLR and the RS pointer allows for three extra pushes that may occur (in a worst case) prior to entering the stack overflow service routine. These pushes would be:

1. the push causing the initial overflow
2. a possible push operation while IV_9 is output and
3. the IR_9 return address push.

See Interrupts, 1.4.3, and Three Stack Pushes on Stack Overflow (appendix 4.2.5) for more details.

The SLR is only 4 bits wide and is compared to the 4 MS bits of the 6-bit RAM address. Therefore, stack limits may only be set at integer multiples of 2^2, i.e., RAM locations 0, 4, 8, 12, ..., 60. The SLR is right-filled the additional two bits with zeros or ones, depending upon the direction of the push being performed (00 for SS pushes and 11 for RS pushes, see Instruction Set Description — SLRIVP, 2.5). In the cycle following a stack overflow, the highest priority interrupt vector (IRV_9 (also used for trapping, see I/R Pin, 1.7) is output. To determine the cause of this interrupt, both SS and RS pointers must be tested in the first several cycles of the service routine. Prior to returning from the overflow interrupt routine, the SLRIVP instruction must be executed, to clear the calling IR_9 from the interrupt latch.

1.4.3 Interrupts

The ADSP-1401 processes eight external and two internal interrupts. All external interrupts are level sensitive (positive logic; see IR Latch, this section) and are processed by the interrupt logic block. The block elements (see Figure 4) are comprised of an interrupt de-multiplexer followed by an interrupt latch, masking logic and priority decoder for selecting the most urgent interrupt (IR_9 having the highest priority, and IR_0 the lowest), and special one-shot to override the address multiplexer with the interrupt vector (IV_{9-0}) on the cycle following the interrupt request.

The external interrupts (IR_{8-1}) may be used for any purpose, however, unused inputs *must not* be left floating (i.e., tie them to logic LO so as to preclude the associated interrupt). Two additional interrupts which are internal are reserved for stack overflow — IR_9 (see Stack Limit Register and Stack Overflow, 1.4.2) and counter underflow — IR_0 (see Counters, 1.4.1). See Counters (1.4.1) for implications of using IR_0 for other than writable control store downloading.

Interrupt vectors are always output (assuming interrupts are enabled and the associated interrupt is not masked) on the cycle immediately following the acceptance of the interrupt request. Contextual saves (stacking and storing) should be made immediately upon entering the interrupt service routine and restored immediately prior to its exit.

Up to four external interrupts may be connected directly to the external interrupt pins, $EXIR_{4-1}$, and are treated as interrupts IR_{8-5}, respectively. Lower priority interrupts, IR_{4-1}, must be masked out in this case.

Up to eight external interrupts may be accommodated using time-division multiplexing. An external 2:1 multiplexer reduces the eight external interrupts to two groups of four (see Figure 3). An internal de-multiplexer automatically restores the external interrupts back to eight.

The interrupt vector file may be directly read and written via the data bus with the aid of the Interrupt Vector Pointer (see Instruction Set Description, Interrupts, 2.5).

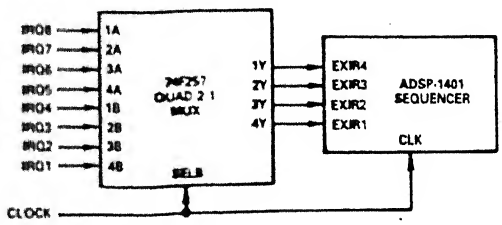


Figure 3 Expanding External Interrupts

IR Latch
Interrupt requests IR_{8-5} are latched during the first half-cycle (clock HI), while IR_{4-1} are latched during the second half-cycle (clock LO). Once latched, external interrupt requests are held until processed, even if the external request signal goes away. This latching technique allows removal of external interrupt sources after they have been recognized by the sequencer.

Latched user interrupt requests (IR_{8-1}) are held until: i) the interrupt is processed and a "Return from Interrupt" (RTNIR) instruction is executed; ii) the interrupt service routine executes a "Clear Current Interrupt" instruction (allowing nested interrupts), or, iii) a "Clear All Interrupts" instruction is executed. Reserved interrupts (IR_9 and IR_0) are cleared from the interrupt latch by utilizing the SLRIVP and CLRS instructions, respectively. See Internal IR Control Logic (1.4.3) for details.

The user may bypass the interrupt latch with the "Select Transparent Interrupts" (STIR) instruction (setting status register bit SR_0). In the transparent mode, the interrupting device must assert the interrupt request until the interrupt service routine resets the request source.

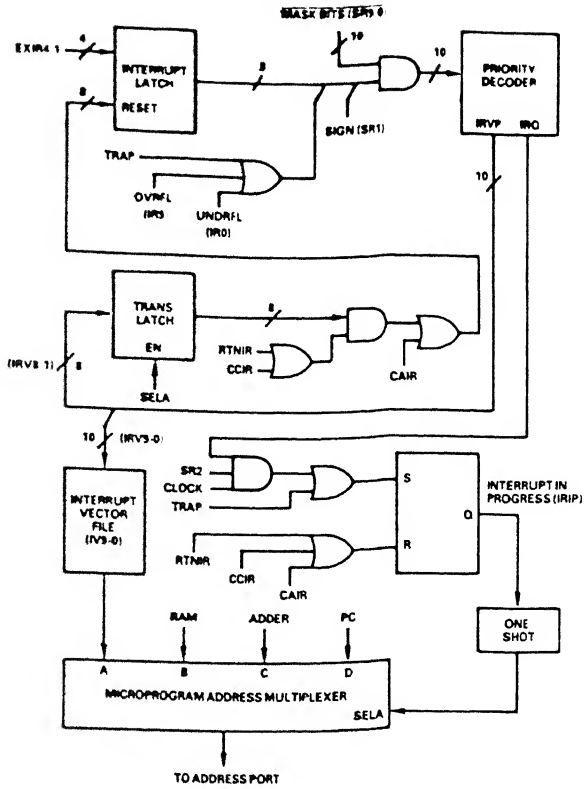


Figure 4. Internal Interrupt Control Logic

IR Mask

All ten interrupts may be independently masked using status register bits SR_{15-6} (corresponding to interrupts IR_{9-0}). Setting a particular mask bit prevents the interrupt from being executed. Note that the status register may be read or written via the Data port, and also pushed and popped to/from the subroutine stack, allowing nesting and servicing of interrupts in any desired order (see Internal IR Control Logic, 1.4.3; and Status Register, 1.4.4).

Two instructions allow bitwise clearing or setting of the interrupt mask. "IR Mask Bit Clear" (IRMBC) will clear those mask bits for which the corresponding data bits (D_{15-6}, as applied to IR_{9-0}) are set, while "IR Mask Bit Set" (IRMBS) will set those mask bits for which the corresponding data bits are set. In both cases, zeros in the data field will preserve the corresponding mask bit. See Instruction Set Description – Status Register, 2.3.

IR Priority Decoder

Unmasked interrupts are passed to the priority decoder which determines the most urgent, valid interrupt and generates an internal Interrupt Request Signal (IRS). The corresponding vector is then fetched from the interrupt vector file and passed to the address port.

Minimum IR Servicing Requirements

Interrupt vectors are output on the cycle following the acceptance of an interrupt request. Interrupt jumps differ from subroutine jumps in that subroutine jumps push the return address in the *same* cycle as the jump address is output, whereas interrupt return addresses are not pushed until the *following* cycle. This is because the instruction executing while the interrupt vector is output may be utilizing RAM and must complete its execution prior to pushing the interrupt return address. Thus, the PC (interrupt return address) is pushed automatically in the first cycle of the interrupt service routine, i.e., *the cycle following the interrupt request acceptance.*

For this reason, the first instruction of any interrupt service routine is always ignored; it *must* be a no-op (CONT). Note that a minimum interrupt service routine would be a CONT followed by a RTNIR.

Internal IR Control Logic

The interrupt enable bit of the status register, SR_2, *must* be set for interrupt servicing to occur. Interrupt servicing may be inhibited by clearing this bit, although external interrupt requests will continue to be latched.

Only one interrupt is ever active at a time. Additional interrupts are "locked out" by an internal "Interrupt In Progress" signal (IRIP) during interrupt servicing (except for TRAP), although they continue to be latched. The IRIP signal is automatically reset upon the "Return from Interrupt" (RTNIR) instruction which pops the return address from the subroutine stack to the PC.

Normally, multiple interrupts are accumulated in the interrupt latch. Whenever a valid interrupt is pending, the internal signal "Interrupt Request" (IRQ) is asserted. Upon each RTNIR, the highest priority, unmasked, pending interrupt is serviced.

Nested interrupts are supported with two instructions: "Clear Current Interrupt" (CCIR) or "Clear All Interrupts" (CAIR). The CCIR instruction clears the IRIP signal and interrupt latch bit for the interrupt in progress. This action re-enables interrupting, relegating the interrupt in progress to a subroutine status. If an external interrupt is pending, the associated IR vector will be output on the cycle following CCIR. To cancel all pending interrupt requests, the CAIR instruction clears the IRIP signal and the entire interrupt latch.

Normally, it is good practice to convert interrupts to subroutines. This can be done by executing the "Clear Current Interrupt" (CCIR) instruction (resetting IRIP) and should be done as early as possible in the interrupt service routine. There are two reasons for changing the status of an interrupt to that of a subroutine. Firstly, if IRIP is allowed to remain active throughout the interrupt service routine, then the occurrence of either internal interrupt (stack overflow or counter underflow, IR_9 or IR_0, respectively) will remain undetected until the current interrupt concludes; the user will be unaware of these interrupt requests.

When using the TRAP capability (see TTR Pin, 1.7), there is a second reason to clear IRIP. Because TRAP must have the highest priority, interrupt IR_0 (when invoked by a TRAP request) is not locked out by IRIP. This allows TRAP to displace an interrupt in progress, but also means that upon completion of the trap service routine, IRIP will be cleared by the RTNIR instruction, re-enabling interrupting in spite of the incomplete interrupt which TRAP displaced.

Either of these instructions (CCIR or CAIR) require an "extra" cycle before a pending interrupt vector may be output. A typical scenario being an interrupt in progress, IR_n (containing a CCIR instruction), with a interrupt pending, IR_m:

CCIR Example

µCode Location	Instruction Executing	Output Address	Comments
n	IR_n Routine	n + 1	IR_m Pending
n + 1	CCIR	n + 2	Clear IRIP
n + 2	IR_m Routine	IV_m	IR_m Recognized
IV_m	IR_m Routine	IV_m + 1	. . .

1.4.4 Status Register

The ADSP-1401 has a 16-bit status register for storing various operational modes. The ten MS bits of this register (SR_{15-6}) comprise the interrupt mask for interrupts IR_{9-0}, respectively. The remaining six LS bits (SR_{5-0}) control the operational modes as shown below.

Status Register Bit Assignments

Bit#	Function (HI/LO)
SR_{15}	IR_9 Mask Bit
.	.
.	.
.	.
SR_6	IR_0 Mask Bit
$SR_{5,4}$	Relative Jump Width Selection: '00' = 16-bit relative address width '01' = 8-bit width '10' = IHC Mode (8-bit width) '11' = 12-bit width
SR_3	Select GSP/LSP
SR_2	Enable/Disable Interrupts
SR_1	Set/Clear Sign Bit
SR_0	Select Transparent/Latched Interrupts

The status register can be directly read and written via the data port and also pushed and popped to/from the subroutine stack. In addition, status register bits SR_{15-6} (the interrupt mask) may be bitwise cleared or set with dedicated instructions. See: Instruction Set Description – Status Register, 2.3; and Interrupts – IR Mask, 1.4.3.

1.5 Clock

The input clock employs both HI and LO levels to control the various transparent latches throughout the device. Generally, the clock should be symmetric; however, in some instances the clock may be stretched during the second half-cycle (LO) to accommodate unusual circumstances such as a cache memory miss (see: TTR Pin – Trap, 1.7).

1.6 External Flag

The external flag input may be used to control conditional instructions. FLAG is latched similarly to instructions (latched during clock HI and transparent during clock LO), but requires less setup time. Two instructions make explicit use of FLAG as their condition (JPCOF and JPCNF), while others employ a condition mode selection (UNCONDITIONAL, NOT FLAG, FLAG, or SIGN; see Instruction Set Description, 2.0, to be specified as part of their opcode.

1.7 TTR Pin (Trap, Three-State and Reset)

The Trap, Three-State and Reset pin (TTR) is a time-multiplexed, three-purpose pin used to

- provide program trap capability
- control the address port output drivers and
- reset the ADSP-1401.

If the TTR pin is held HI for an entire cycle, the RESET sequence begins and TTR must be held HI for at least two more complete cycles (RESET requires three cycles to complete). If trap and three-state control capabilities are also needed, the combination of the 1401's internal circuits and the external circuitry shown in Figure 5 can be used to effectively time-multiplex the TTR pin.

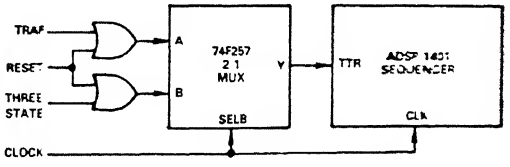


Figure 5. External Logic for TTR Pin

Trap

For a trap to occur, the TTR pin must be asserted during clock LO only. The primary reason to invoke a trap is in support of cache memory systems, or in case of system emergencies. Cache memory systems generally utilize a large microcode memory space, of which only a small area (that currently under execution) is comprised of high-speed RAM (the balance consisting of slower, less costly memory). The high-speed RAM is directly accessible by the sequencer, whereas the bulk of (slow) memory is usually accessible indirectly (via a cache memory controller which controls downloads of code to the cache memory area).

In a cache-based system, microcode is generally executed from the high-speed cache. If an access is attempted to code not resident in the cache area, the cache memory controller must detect the discrepancy and generate an exception to the access (a "cache miss"). Then, the missing code segment must be downloaded to the cache memory area (see Instruction Set Description - Writeable Control Store, 2.7).

When a cache miss occurs, the cache memory control logic asserts the TTR pin while stretching the system clock LO. Upon detecting the trap request, the sequencer immediately generates the highest priority interrupt, IR_9, replacing the current address (that causing the cache miss). The cache miss address is pushed on the subroutine stack and popped after the interrupt service routine has reloaded the cache area with the missing code segment.

The trap interrupt differs from the standard interrupt protocol in three ways:

1. The interrupt vector, IV_9, is output asynchronously, i.e., it occurs t_{TRAP} after asserting the Trap signal and must occur *before* the next cycle. To accomplish this, a clock stretch cycle may be needed to allow enough time to fetch the new instruction.
2. The *current address* is pushed onto the SS for later restoration (after the cache miss is resolved), whereas standard interrupts push the *current address + 1*.
3. Trap interrupts *cannot* be masked or disabled. Note that if IR_9 is also used for stack overflow and underflow, the service routine must discriminate which actually occurred.

Caution: because trapping is asynchronous, spikes on the TTR pin wider than 3ns during clock LO may initiate inadvertent trapping.

Three-State

The address port is placed in a high impedance state when the TTR pin is HI during clock HI and LO during clock LO. The TTR signal is latched during clock LO and transparent during clock HI. This facilitates full cycle, three-state control. (Note that the IDLE instruction can also place the address port in a high-impedance state.)

Reset

The TTR pin may be used to initialize the ADSP-1401 by asserting it (HI for both clock phases) for at least three full cycles. Use of the reset operation alone does not require the multiplexing described above. However, if the trap and/or three-state controls are also needed, they must not occur in the same cycle (this would be an abnormal situation), as this constitutes a reset. The RESET signal forces a zero output address, places the data port in the high-impedance state, and resets internal registers as follows:

Sequencer Status after RESET Operation

Parameter	Reset Condition
Program Counter	µCode Location 0000_{16}
Subroutine Stack Pointer (SSP)	RAM Location 00_{10}
Local Stack Pointer (LSP)	Undefined
Global Stack Pointer (GSP)	Undefined
Stack Limit Register (SLR)	RAM Location 32_{10}
RAM Data	No Change
Counters	No Change
Interrupt Mask (SR_{15-6})	All Bits to '0' (Unmasked)
Interrupt Vector File	No Change
Interrupt Vector Pointer (IVP)	Undefined
SR_{5-4}	'00' (16-Bit Relative Offsets)
SR_3	'0' (LSP Selected)
SR_2	'0' (Interrupts Disabled)
SR_1	'0' (Sign Bit Cleared)
SR_0	'0' (Latched Interrupt Mode)
Writeable Control Store Mode	Cleared

NOTE:
The first instruction (microcode location 0000_{16}) must be a "CONT".

2.0 INSTRUCTION SET DESCRIPTION

The instruction set is divided into seven categories pertaining to generic operation (see data sheet outline or Mnemonics and Opcodes, 4.5).

Several instructions employ two instruction bits (I_1 and I_0) to specify a counter (C_{3-0}) and or a local register (R_{3-0}, relative to the RSP, as arguments. Nine of the conditional instructions use another two instruction bits (I_3 and I_2) to select one of the four condition modes:

'00'	UNCONDITIONAL
'01'	NOT FLAG
'10'	FLAG
'11'	SIGN

The sign bit of the status register, SR_1, may also be used to (implicitly or explicitly) store an external condition. This is useful if the condition results from an operation performed in the middle of a loop, but is not tested until the end, the loop is exited with an "If Sign Jump" instruction. Recall that any subsequent counter operations will overwrite SR_1.

2.1 Jump and Branch Instructions

Jump and branch instructions provide flow control of microcode execution, offering three-way branches, jumps, subroutine calls, returns, and addressing mode selection (see Figure 6). These instructions support conditional control, allowing addressing from the register stack, the data port, or the indirect jump address space in the RAM. Generally, they are of the form:

If Condition: Do Operation; Else, Continue.

JPCOF IFFLAG: JUMP PC

The address is not incremented while the flag is at a logic HI, i.e., PC<=PC. If the flag is LO, the next address is (PC+1).

JPCNF IFNOT FLAG: JUMP PC

The address is not incremented while the flag is at a logic LO, i.e., PC<=PC. If the flag is HI, the next address is (PC+1).

JTWO IFCONDITION: JUMP PC+2

If the condition specified is met, this instruction causes the next sequential microprogram address to be skipped. This instruction allows single instruction bypassing or interleaving without need to provide explicit addressing.

JDA IFCONDITION: JUMP DATA, ABSOLUTE

If the specified condition is met, this instruction causes a jump to the absolute address at the data port. If the condition is not met, the next sequential instruction will be executed.

JSR IF CONDITION: JUMP SUBROUTINE, RELATIVE

If the condition specified is met, the address at the data port is added to the PC and output (jump distance is offset plus one) and the PC is pushed onto the subroutine stack. The offset width is determined by the address width selection (8, 12, or 16-bits). If the condition is not met, the next sequential instruction will be executed.

RTN IF CONDITION: RETURN FROM SUBROUTINE

This instruction is used to return from subroutines. If the condition specified is met, the subroutine stack is POPped, which outputs the return address and decrements the SSP. If the condition is not met, the next sequential instruction will be executed.

BRANCH IF SIGN OF C_i: JUMP R_i, $C_i <= C_i - 1$;
ELSE, IF CONDITION:
JUMP DATA, $C_i <= C_i - 1$;
ELSE, $C_i <= C_i - 1$ (COND ≠ SIGN)

This instruction implements a three-way branch with the address source from the data port, register R_i, or the PC. The instruction first tests the sign bit of the counter C_i; if negative, the output address is given by R_i, i.e., RSP + i. If the sign was not true, but the specified condition is true, the address source is the data port. If the sign was not true and the condition is not met, the next sequential instruction is executed.

The counter and the register use the same subscript value i. The counter is *always* decremented. Note that this instruction uses only absolute data addresses; relative addressing is not available with the three-way branch instruction.

2.2 Stack Operations

Subroutine Stack

Subroutine Stack Pointer (SSP) instructions are used for maintaining the subroutine stack. These instructions may also be used to upload or download the entire RAM for examination, stack expansion or context switches.

PSDSS PUSH DATA ONTO SS

Increments the stack pointer and then loads the RAM location specified by the SSP with the data at the data port.

PPSSD POP SS TO DATA PORT

Transfers the contents of the stack location given by the stack pointer to the data port and decrements the stack pointer.

WRSSP WRITE SSP

Loads the SSP with bits D_{5-0} of the data port.

RDSSP READ SSP

Read the 6-bit subroutine stack pointer. This allows the value of the stack pointer to be saved or examined. Bits D_{5-0} of the data port correspond to bits 5-0 of the SSP. The 10 MSB's of the data port (D_{15-6}) are undefined.

DSSP DECREMENT SSP

Decrements the stack pointer without reading.

Register Stack

Register Stack Pointer (RSP) instructions are used to upload and download the entire RAM for initialization, examination, or context switching and to maintain the RAM space allocated to local and global jump registers. As previously discussed, register stack instructions refer to either the Local Stack Pointer (LSP) or the Global Stack Pointer (GSP), depending upon the status register (SR_3). If SR_3 is LO, register stack instructions pertain to the LSP. If SR_3 is HI, register stack instructions pertain to the GSP.

SGSP SELECT GSP

Select the Global Register Stack Pointer. Set Status bit SR_3 (HI).

SLSP SELECT LSP

Select the Local Register Stack Pointer. Clear Status bit SR_3 (LO).

RDRSP READ RSP

Transfers the RSP to the data port bits D_{5-0} for examination or storage. The 10 MSBs (D_{15-6}) of the D port are undefined.

WRRSP WRITE RSP

Preload the selected RSP (LSP or GSP) with bits D_{5-0} of the data port.

PSPC PUSH PC ONTO RS

Decrements the RSP and writes the PC to the register stack. This instruction may be used to set up a JRC loop (IF CONDITION: JUMP R_i = PC).

PSGSP PUSH GSP ONTO SS

Increment the SSP and write the GSP onto the subroutine stack.

PPGSP POP GSP FROM SS

Write the subroutine stack to the GSP and decrement the SSP.

PSDRS PUSH DATA ONTO RS

Decrement the RSP and then write the data at the data port into the location specified by the updated RSP.

PPRSD POP RS TO DATA PORT

Transfers RAM data pointed to by the RSP to the data port and then increments the RSP.

AIRSP ADD i TO RSP

Add i to the register stack pointer. Note that i = 0, 1, 2, or 3 in this instruction corresponds to 4, 1, 2, or 3, respectively. This instruction effectively removes up to four registers from the stack.

S1RSP SUBTRACT ONE FROM RSP

Subtract 1 from the RSP without a write. This instruction is used to modify the RSP without explicitly reloading it.

S4RSP SUBTRACT FOUR FROM RSP

Subtract four from the RSP without a write. This instruction may be used to modify the RSP without explicitly reloading it.

2.3 Status Register Operations

The status register bits, SR_{15-0}, contain ten mask bits, SR_{15-6}, for masking interrupts IR_{9-0}, and six control bits, SR_{5-0} (see

Bidirectional Data Port, 1.4). The entire status register can be read or written via the data port, or pushed or popped to/from the subroutine stack. Upon RESET, the entire status register is initialized to zero.

RDSR READ SR

The entire status register (SR_{15-0}) is output over the data port (D_{15-0}).

WRSR WRITE SR

Write the data port (D_{15-0}) to the status register (SR_{15-0}).

PSSR PUSH SR ONTO SS

Increment the SSP and then write the status register to the subroutine stack.

PPSR POP SR FROM SS

The top of the subroutine stack is written into the status register, and then the SSP is decremented.

2.4 Counter Operations

Counters may be pushed and popped to/from the subroutine stack or loaded directly from the data port. The counters may be read externally by pushing the counters onto the subroutine stack then popping the subroutine stack to the data port. The device has four counters, denoted C_i, which are indexed by the two LSBs of the instruction.

If a jump is required *after* N events (until sign), the counter should be loaded with two less than the number of events desired (N - 2). If a jump is required *for* N events (while sign), the counter is loaded with $2^{15} + N - 2 = 8000_{16} + N - 2$.

Care must be taken when using the counter underflow interrupt (IR_0, see 1.4.3) to clear the sign bit *before* the IR_0 mask bit is cleared.

WRCNTR WRITE C_i

Write to the selected counter, C_i, from the data port.

CLRS CLEAR SIGN BIT

Clear status register bit SR_1.

SETS SET SIGN BIT

Set status register bit SR_1.

PSCNTR PUSH C_i ONTO SS

Increment the SSP and write the specified counter onto the subroutine stack.

PPCNTR POP C_i FROM SS

Transfer the data from the subroutine stack to the counter specified by the instruction, then decrement the SSP.

DCCNTR DECREMENT C_i

Unconditionally decrement counter C_i.

IFCDEC IF CONDITION: DECREMENT C_0

Decrement counter C_0 on condition. If the sign condition is selected, the sign is taken from the status register bit SR_1, rather than from the counter sign (which normally provides the sign condition).

Normally, if the counter underflow interrupt (IR_0) is enabled, it is activated by the counter sign bit going HI. However, if IFCDEC is used to decrement C_0, the IR_0 interrupt is activated by the SR_1 bit, rather than the sign bit of C_0. Since the SR_1 bit goes HI only after C_0 has underflowed, IFCDEC must be executed once more after the C_0 underflow to generate the IR_0 interrupt. Alternatively, the preloaded value of C_0 may be reduced by one.

2.5 Interrupt Control

Detailed interrupt operation is described in the Interrupts section (1.4.3). Here, specific interrupt operations such as interrupt clearing, IRV read/write, interrupt mask manipulation, etc., are described.

CCIR CLEAR CURRENT INTERRUPT

Allows nesting of user interrupts IR_{8-1} on subsequent instructions by clearing both the interrupt latch bit currently being serviced and the interrupt in progress signal (IRIP), re-enabling interrupts. If an external interrupt is pending, the associated IR vector will not be output until the cycle following CCIR. Internal interrupts (IR_9 and IR_0) are *not* cleared by CCIR and must be explicitly cleared through the SLRIVP and CLRS instructions, respectively.

CAIR CLEAR ALL INTERRUPTS

Clears external interrupt latches IR_{8-1}, and re-enables the interrupt interface (IRIP cleared LO). The next sequential instruction will be executed prior to the jump to a pending interrupt. Internal interrupts (IR_9 and IR_0) are *not* cleared by CAIR and must be explicitly cleared through the SLRIVP and CLRS instructions, respectively.

RTNIR RETURN FROM INTERRUPT

Clears the current interrupt latch for IR_{8-1}, re-enables interrupts (IRIP cleared LO), and pops the return address from the subroutine stack. The next sequential instruction will be executed prior to the jump to a pending interrupt routine. Internal interrupts are *not* cleared and the IR_9 and IR_0 interrupt latches must be cleared explicitly through the SLRIVP and CLRS instructions, respectively.

RDIV READ IRV AND INCREMENT IVP

Outputs the interrupt vector currently pointed to by IVP to the data port and then increments the IVP. Interrupts should be disabled when writing or reading interrupt vectors.

WRIV WRITE IRV AND INCREMENT IVP

Writes the interrupt vector currently pointed to by the IVP from the data port and then increments the IVP. Interrupts should be disabled when writing or reading interrupt vectors.

IRMBC IR MASK BITWISE CLEAR

Allows selected IR mask bits to be cleared. Data port bits D_{15-6} are applied to status register bits SR15-6 (corresponding to mask bits for IR_{9-0}). Those data bits which are HI will clear the mask bit, while those data bits which are LO will leave the mask bit intact. Data port bits D_{5-0} are ignored.

Bidirectional Data Port, 1.4). The entire status register can be read or written via the data port, or pushed or popped to/from the subroutine stack. Upon RESET, the entire status register is initialized to zero.

RDSR READ SR

The entire status register (SR_{15-0}) is output over the data port (D_{15-0}).

WRSR WRITE SR

Write the data port (D_{15-0}) to the status register (SR_{15-0}).

PSSR PUSH SR ONTO SS

Increment the SSP and then write the status register to the subroutine stack.

PPSR POP SR FROM SS

The top of the subroutine stack is written into the status register, and then the SSP is decremented.

2.4 Counter Operations

Counters may be pushed and popped to/from the subroutine stack or loaded directly from the data port. The counters may be read externally by pushing the counters onto the subroutine stack then popping the subroutine stack to the data port. The device has four counters, denoted C_i, which are indexed by the two LSBs of the instruction.

If a jump is required *after* N events (until sign), the counter should be loaded with two less than the number of events desired (N - 2). If a jump is required *for* N events (while sign), the counter is loaded with $2^{15} + N - 2 = 8000_{16} + N - 2$.

Care must be taken when using the counter underflow interrupt (IR_0, see 1.4.3) to clear the sign bit *before* the IR_0 mask bit is cleared.

WRCNTR WRITE C_i

Write to the selected counter, C_i, from the data port.

CLRS CLEAR SIGN BIT

Clear status register bit SR_1.

SETS SET SIGN BIT

Set status register bit SR_1.

PSCNTR PUSH C_i ONTO SS

Increment the SSP and write the specified counter onto the subroutine stack.

PPCNTR POP C_i FROM SS

Transfer the data from the subroutine stack to the counter specified by the instruction, then decrement the SSP.

DCCNTR DECREMENT C_i

Unconditionally decrement counter C_i.

IFCDEC IF CONDITION: DECREMENT C_0

Decrement counter C_0 on condition. If the sign condition is selected, the sign is taken from the status register bit SR_1, rather than from the counter sign (which normally provides the sign condition).

Normally, if the counter underflow interrupt (IR_0) is enabled, it is activated by the counter sign bit going HI. However, if IFCDEC is used to decrement C_0, the IR_0 interrupt is activated by the SR_1 bit, rather than the sign bit of C_0. Since the SR_1 bit goes HI only after C_0 has underflowed, IFCDEC must be executed once more after the C_0 underflow to generate the IR_0 interrupt. Alternatively, the preloaded value of C_0 may be reduced by one.

2.5 Interrupt Control

Detailed interrupt operation is described in the Interrupts section (1.4.3). Here, specific interrupt operations such as interrupt clearing, IRV read/write, interrupt mask manipulation, etc., are described.

CCIR CLEAR CURRENT INTERRUPT

Allows nesting of user interrupts IR_{8-1} on subsequent instructions by clearing both the interrupt latch bit currently being serviced and the interrupt in progress signal (IRIP), re-enabling interrupts. If an external interrupt is pending, the associated IR vector will not be output until the cycle following CCIR. Internal interrupts (IR_9 and IR_0) are *not* cleared by CCIR and must be explicitly cleared through the SLRIVP and CLRS instructions, respectively.

CAIR CLEAR ALL INTERRUPTS

Clears external interrupt latches IR_{8-1}, and re-enables the interrupt interface (IRIP cleared LO). The next sequential instruction will be executed prior to the jump to a pending interrupt. Internal interrupts (IR_9 and IR_0) are *not* cleared by CAIR and must be explicitly cleared through the SLRIVP and CLRS instructions, respectively.

RTNIR RETURN FROM INTERRUPT

Clears the current interrupt latch for IR_{8-1}, re-enables interrupts (IRIP cleared LO), and pops the return address from the subroutine stack. The next sequential instruction will be executed prior to the jump to a pending interrupt routine. Internal interrupts are *not* cleared and the IR_9 and IR_0 interrupt latches must be cleared explicitly through the SLRIVP and CLRS instructions, respectively.

RDIV READ IRV AND INCREMENT IVP

Outputs the interrupt vector currently pointed to by IVP to the data port and then increments the IVP. Interrupts should be disabled when writing or reading interrupt vectors.

WRIV WRITE IRV AND INCREMENT IVP

Writes the interrupt vector currently pointed to by the IVP from the data port and then increments the IVP. Interrupts should be disabled when writing or reading interrupt vectors.

IRMBC IR MASK BITWISE CLEAR

Allows selected IR mask bits to be cleared. Data port bits D_{15-6} are applied to status register bits SR15-6 (corresponding to mask bits for IR_{9-0}). Those data bits which are HI will clear the mask bit, while those data bits which are LO will leave the mask bit intact. Data port bits D_{5-0} are ignored.

IRMBS IR MASK BITWISE SET

Allows selected IR mask bits to be set. Data port bits D_{15-6} are applied to status register bits SR_{15-8} (corresponding to mask bits for IR_{9-0}). Those data bits which are HI will set the mask bit, while those data bits which are LO will leave the mask bit intact. Data port bits D_{5-0} are ignored.

DISIR DISABLE INTERRUPTS

Disables the execution of all further interrupts by clearing the enable interrupt flag (SR_2). External interrupts continue to be latched.

ENAIR ENABLE INTERRUPTS

Enables execution of interrupts by setting the enable interrupt flag (SR_2).

SLIR SELECT LATCHED INTERRUPTS

Places the interrupt request latches in the latched mode for interrupts IR_{8-1} (SR_0 LO). Interrupts are latched if they are valid at the appropriate clock edge. Interrupts IR_{8-5} are latched at the positive going clock edge while IR_{4-1} are latched at the negative going clock edge.

STIR SELECT TRANSPARENT INTERRUPTS

Places the interrupt request latches in the transparent mode (SR_0 HI) for interrupts IR_{8-1}. The interrupt request is only valid while the external interrupt inputs are high. Interrupts are still processed on the next cycle, so long as they meet the minimum interrupt setup specification. Note that selecting transparent interrupting will clear any pending interrupts stored in the interrupt latch.

SLRIVP WRITE SLR WITH D_{3-2}, AND IVP WITH D_{15-12}

Loads the 4-bit stack limit register (SLR) and the 4-bit interrupt vector pointer (IVP) from the data port. This instruction also clears the stack overflow interrupt request IR_9.

For stack overflow detection, the active 6-bit stack pointer (SSP, LSP or GSP) is compared to a 6-bit word comprised of the 4-bit SLR (MSBs) and the two LSBs determined by the instruction type, as follows:

'00' for subroutine stack push (PSDSS); or,
'11' for register stack push (PSDRS).

For example, if a stack limit of 36_{10} and positioning of the IVP at IRV_7 is desired, the value '0111xxxxxx1001xx' is provided at the data port. Note that the SLR and IVP cannot be read.

The interrupt vector pointer (IVP) addresses the vector file for reading or writing interrupt vectors. To write interrupt vectors IRV_{9-0}, the IVP must first be initialized by SLRIVP. The WRIV instruction (see above) is then used to write the interrupt vector pointed to by the IVP, which is then incremented automatically.

2.6 Relative Address Width Controls

The width control instructions allow reduction of microcode when Jump Data Relative and Jump Subroutine Relative instructions need less than the full, 16-bit range. Use these instructions to sign extend the 8, 12 or 16-bit wide jump data presented at the data port. The jump width may be selected by the explicit instructions or by directly setting the status register bits SR_{5-4} as described below. Any of these three instructions will reset the Instruction Hold Control mode (see Misc. Instructions – IHC, 2.7).

Note that selection of 8-bit width can be made with or without IHC. For all relative jumps, the jump distance is the offset + 1.

REL16 SELECT 16-BIT RELATIVE JUMPS

Select the 16-bit relative jump. This adds D_{15-0} at the data port to the PC to obtain the jump address. The status bits SR_{5-4} are set to '00'.

REL12 SELECT 12-BIT RELATIVE JUMPS

Selects the jump data from D_{11-0}. The offset is sign-extended allowing relative jumps in the range +2047 to −2048. The status bits SR_{5-4} are set to '11'.

REL8 SELECT 8-BIT RELATIVE JUMPS

Selects the jump data from D_{7-0}. The offset is sign-extended allowing relative jumps in the range +127 to −128. The status bits SR_{5-4} are set to '01'.

2.7 Miscellaneous Instructions

CONT CONTINUE

Increment and output the next location in microcode memory without any other changes. Allows straight line microcode execution.

IDLE DISABLE OUTPUTS AND JUMP PC

Places the address port into the high-impedance state, inhibiting program counter (PC) increments. Useful in applications where multiple sequencers share a common microcode address bus.

This instruction causes the ADSP-1401 to behave as if the clock had stopped. The IDLE instruction may be latched internally by using IHC, freeing microcode for use by another device.

External interrupt requests must be inhibited during IDLE. If interrupts are not inhibited, the ADSP-1401 will attempt to process an interrupt that goes active. However, it will be unable to output an interrupt vector because the IDLE instruction places the address port in the high-impedance state; more importantly, it will set its IRIP flag, which will inhibit further interrupt processing even after the IDLE state is exited.

Interrupts can be inhibited using the interrupt mask or the DISIR instruction. While inhibited, interrupt requests will still be latched in the interrupt latch.

IHC ENABLE INSTRUCTION HOLD CONTROL

Sets SR_{5-4} to '10' and redefines the function of IR_1 to allow a subsequent instruction to be held for repeated execution, regardless of the instruction port. Use of the IHC mode requires that the mask bit for IR_1 be set. See Instruction Hold Control, appendix 4.1 for more details.

While in the IHC mode, asserting IR_1 HI (prior to the second half-cycle of any instruction) will hold that instruction and disable *all* interrupts (although they continue to be latched) until IR_1 is brought LO again (again, prior to the second half-cycle of any instruction).

It is recommended that IR_1 be dedicated to control of the IHC mode (if needed). However, if it must also be used for subsequent interrupting, then the CAIR instruction should be executed before unmasking IR_1 (to clear the interrupt request resulting from use of IR_1 as the IHC control).

Use of IHC is constrained to 8 bit relative addressing (see Relative Address Width Controls, 2.6) and clearing IHC is accomplished by executing any of the relative address width control instructions (changing status register bits SR1-4).

WCS — WRITE CONTROL STORE

Provides sequential addressing during microcode downloads to a RAM based microcode store. The instruction may be interpreted as:

JUMP DATA,
IF FLAG DECREMENT C_0 AND CONTINUE UNTIL INTERRUPTED

Upon initiation of the WCS instruction, the sequencer outputs the address found at the data port (that of the first instruction to be downloaded). The external flag is then used to gate subsequent sequential addressing for the download and decrementing of counter C_0. This action continues until an interrupt is detected (from either a C_0 underflow, externally or the chip is RESET). Instructions at the instruction port are *ignored* during WCS, until the interrupt or reset occurs.

The external flag allows synchronization of an external memory with the sequencer. FLAG should be asserted HI as each new microcode word is made available for writing to microcode memory.

Notes on Using a Writeable Control Store

- If a counter interrupt is desired, counter C_0 must be initialized with *two* less than the length of microcode segment to be downloaded.
- If counter interrupting is to be used to exit the WCS mode, IRV_0 should be unmasked and initialized with the address of the instruction to be executed upon WCS completion (see Interrupts, 1.4.3 for timing).
- Since interrupting is used to exit the WCS mode, the last address downloaded is pushed onto the SS stack as an interrupt return address. However, because it is not actually a return address, the SS should be popped immediately by decrementing the SSP (DSSP) to clear it of this last address.
- Since FLAG is used to gate the download, it should not become active until after the WCS instruction is executed.

See application note "Writeable Control Store using the ADSP-1401".

3.0 SPECIFICATIONS

This section describes the ADSP-1401's performance parameters. The Specifications Table lists the device's relevant electrical and switching characteristics, while Figure 7 presents the corresponding timing diagram.

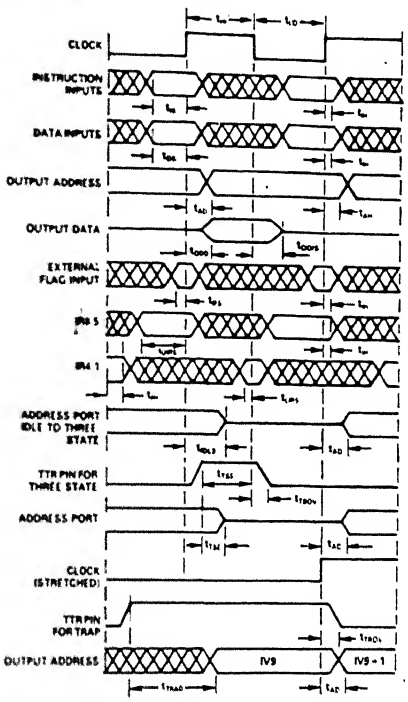


Figure 7. ADSP-1401 Timing Diagram

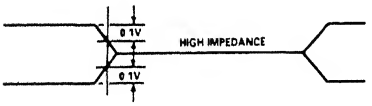


Figure 8. Three-State Reference Levels

ORDERING INFORMATION

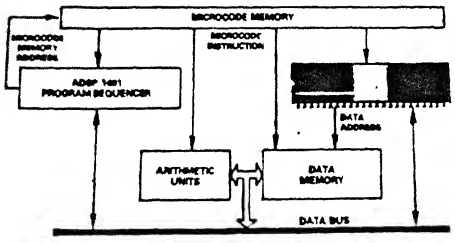
Part Number	Temperature Range	Package	Package Outline
ADSP-1401JN	0 to +70°C	48-Pin Plastic DIP	N-48A
ADSP-1401KN	0 to +70°C	48-Pin Plastic DIP	N-48A
ADSP-1401JP	0 to +70°C	52-Lead PLCC	P-52
ADSP-1401KP	0 to +70°C	52-Lead PLCC	P-52
ADSP-1401JD	0 to +70°C	48-Pin Ceramic DIP	D-48A
ADSP-1401KD	0 to +70°C	48-Pin Ceramic DIP	D-48A
ADSP-1401SD	−55°C to +125°C	48-Pin Ceramic DIP	D-48A
ADSP-1401TD	−55°C to +125°C	48-Pin Ceramic DIP	D-48A
ADSP-1401SD/+	−55°C to +125°C	48-Pin Ceramic DIP	D-48A
ADSP-1401TD/+	−55°C to +125°C	48-Pin Ceramic DIP	D-48A
ADSP-1401SD/883B	−55°C to +125°C	48-Pin Ceramic DIP	D-48A
ADSP-1401TD/883B	−55°C to +125°C	48-Pin Ceramic DIP	D-48A

ANALOG DEVICES

Word-Slice Address Generator

ADSP-1410

FEATURES
16-Bit Addresses with Higher Precision Options
30ns Address Output Delay @ 11.1MHz Operation
Look-Ahead™ Pipeline
Versatile Addressing Hardware
  30 16-Bit Registers
  16-Bit ALU with Left/Right Shift & Carry I/O
  Comparator
  Bit Reverser
Dual Ports
Powerful Single-Cycle Looping Instructions
375mW Maximum Power Dissipation with CMOS Technology
48-Pin DIP, 52-Lead PLCC



WORD-SLICE™ MICROCODED SYSTEM WITH ADSP-1410

GENERAL INFORMATION

The ADSP-1410 is a fast, flexible address generator optimized for digital signal array processors and other high-performance computers. This low-power CMOS device rapidly generates the data memory addresses required by routines such as digital filters, FFTs, matrix operations, and DMAs. With its 16-bit architecture, registers, dual ports, and speed, this Word-Slice™ component improves performance and reduces board space substantially relative to bit-slice solutions.

The ADSP-1410's architecture features a 16-bit ALU, a comparator, and 30 16-bit registers. The registers are organized into four files: sixteen address (R) registers, six offset (B) registers, four compare (C) registers, and four initialization (I) registers.

The ADSP-1410 rapidly executes key address generating operations. In a single instruction cycle, the device can:

- output a 16-bit memory address;
- modify this memory address; and,
- detect when the address value has moved to or beyond a pre-set boundary and conditionally loop back to the top of a circular buffer.

Consequently, circular buffers and modulo addressing for data memories can be implemented without overhead.

The ADSP-1410's 10-bit microcode instructions include commands for looping, register read/writes, internal data transfers, and logical/shift operations. Instructions are normally supplied from an external source. However, an internal Alternate Instruction Register (AIR) can provide the instruction under external control, allowing microcode to be conserved in many applications.

Look-Ahead and Word-Slice are trademarks of Analog Devices, Inc.

The ADSP-1410 has a 16-bit address (Y) port for outputting addresses and a 16-bit data (D) port for I/O between internal and external registers. Also, an internal path allows external data, provided via the D port, to serve as an ALU source and/or to be directly output over the Y port for a DMA capability.

Double-precision (30-bit), single-cycle addressing can be performed by cascading two ADSP-1410's, with the MSB of each chip's D and Y port dedicated to interchip communication. Alternatively, a single AG can provide double-precision addresses at a rate of one per two clock cycles.

The Look-Ahead pipeline eliminates the need for an external microcode pipeline register by internally latching instructions and addresses; microcode bits may be directly routed to the ADSP-1410 from microcode memory. Logically, the Look-Ahead pipeline is split into two halves: the first, located at the instruction (and data) port; and the second, located at the address port. Each half of the pipeline (input vs. output) has a transparent latch which operates out of phase with the other: the address latch is transparent during the first half of the cycle (clock HI), while the input latches (instruction and data) are transparent during the second half of the cycle (clock LO). This complementary arrangement allows new instructions to be decoded (in preparation for the following cycle) while the program address for the current cycle is held steady.

Digital Signal Processing (DSP) and array processing systems require fast, flexible address generation circuitry. An Address Generator (AG) supplies the address of a location in data or coefficient memory. The value residing at the specified address is fetched and fed to an arithmetic unit for processing. The AG must then modify the address pointer in anticipation of the next data fetch. For algorithms that repetitively loop through data buffers, the AG may need to compare the address to a buffer end and conditionally loop back to the top of the buffer. Finally, to maximize throughput, an AG must perform its addressing tasks rapidly and without overhead.

With the ADSP-1410, 16-bit pointers to memory are stored in an address (R) register file. Since an AG must track several pointers concurrently, sixteen R registers, denoted R_n, are provided. If we denote Y as the address port, the operation "$Y \leftarrow R_n$" corresponds to the AG supplying an address from register R_n.

After supplying an address, the AG must update the pointer for the next memory fetch. The updating may be as simple as an increment but, more generally, involves adding or subtracting an arbitrary offset value. Also, algorithms generally access several different offset values. To this end, the AG provides six offset registers, denoted B_m, and can execute in a single-cycle the core operation:

$$Y \leftarrow R_n;\ R_n \leftarrow R_n + B_m.$$

In DSP applications, data arrays are often addressed as circular buffers. That is, when addressing reaches the buffer end, it wraps back to the beginning of the buffer. To implement this looping, the AG compares the supplied address to one of four compare registers, denoted C_i. If the address has moved to or beyond the end of the boundary ($R_n \geq C_i$), the device can transfer an initialization register value, denoted I_i, to the register ($R_n \leftarrow I_i$); otherwise, it is updated in normal fashion ($R_n \leftarrow R_n + B_m$). To minimize overhead, the AG can execute normal updates while also performing conditional re-initializations; again, in one core operation:

$$Y \leftarrow R_n;\ \text{IF } (R_n \geq C_i):\ R_n \leftarrow I_i;\ \text{ELSE } R_n \leftarrow R_n + B_m.$$

Since the above instruction handles the looping required of circular buffer addressing, it is termed a looping instruction. To a large extent, the ADSP-1410's architecture and instruction set revolve around efficient implementation of this instruction. However, many variations of this instruction are supported on the device and spelled out in the following sections.

ADDRESS SOURCES
- Sixteen internal R registers
- External data provided over the D port

OFFSET SOURCES
- Six internal B registers
- Data Port

OFFSET OPERATIONS
- Increment ($R_n \leftarrow R_n + 1$)
- Decrement ($R_n \leftarrow R_n - 1$)
- Add Offset ($R_n \leftarrow R_n + B_m$)
- Subtract Offset ($R_n \leftarrow R_n - B_m$)
- Single-Bit Left Right Shifts
- Logical Operations (AND, OR, XOR)

CONDITIONAL RE-INITIALIZATION
- Independent Inhibit Enable for each of four initialization registers
- Conditional AIR execution (used for true modulo addressing)

OUTPUT UPDATE SEQUENCE
- Normal (Pre-Update) Mode (output the address before update)
- Post-Update Mode (output the address after update)

PRECISION
- Single chip supplies 16-bit addresses
- Two chips cascaded provide 30-bit addresses
- One chip provides 30-bit addresses in two cycles

ADSP-1410 PIN ASSIGNMENTS

PIN NAME	DESCRIPTION
$Y_{15} - Y_0$	The address (Y) output port. In single-chip double-precision mode, the MSB (Y_{15}) indicates whether the supplied address is the MSW or LSW (see Precision Modes). In two-chip double-precision mode, the MSB conveys the carry/shift bit from the Least Significant (LS) to the Most Significant (MS) chip.
$D_{15} - D_0$	The bi-directional data (D) port. In two-chip double-precision addressing mode, the MSB (D_{15}) of this port conveys CMP status from the partner chip.
$I_9 - I_0$	The instruction port.
CMP/Z	A dual function pin. Looping instructions, which compare address register values to compare register values, assert this pin HI to convey CMP status if i) R≥C for positive offsets, or ii) R≤C for negative offsets. Logical/Shift instructions assert this pin HI to convey the ZERO status of the result.
DSEL	Data Select control. Asserting this control HI causes data set up on the data port to substitute for the R value specified in the instruction.
AIR Enable	Alternate Instruction Register control. Asserting this control HI causes the device to execute an instruction stored in the internal AIR, rather than the instruction set up on the instruction port.
CLK	Clock
V_{dd}	+5 Volt Power Supply
GND	Ground

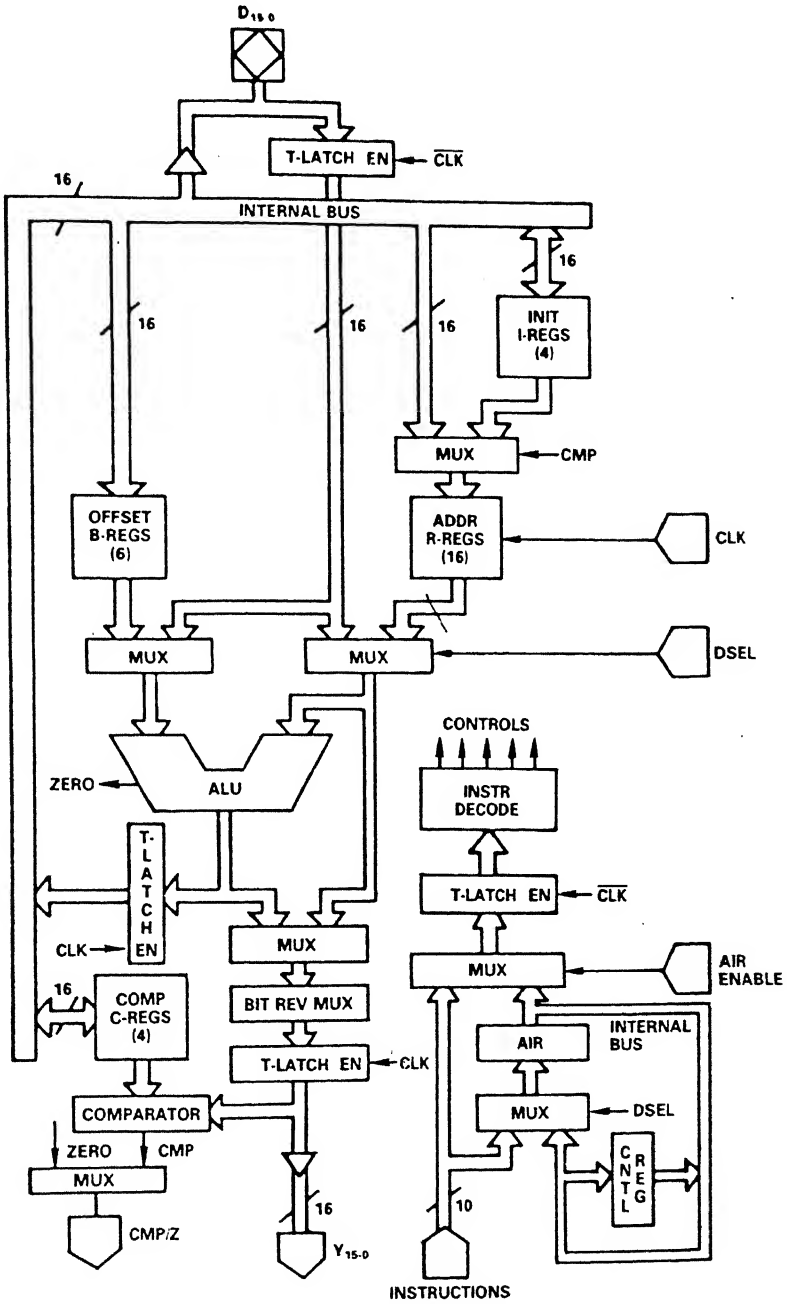


Figure 1. ADSP-1410 Functional Block Diagram

ARCHITECTURE

After discussing the architecture of the ADSP-1410, different operating modes of the ADSP-1410 are detailed, followed by a description of the ADSP-1410's method of operation: including timing concerns and instructions. Brief applications information is then presented, and the data sheet concludes with a section on MNEMONICS AND OPCODES.

The ADSP-1410's architecture (Figure 1) features four register files, an ALU, a Comparator, an Alternate Instruction Register (AIR), and a Control register. External interfaces include a 10-bit instruction port, a 16-bit data (D) and address (Y) ports, a DSEL (Data Select) control pin, an AIR Enable control pin, and a status flag.

Instruction Port

The microcode controlling the ADSP-1410 is supplied over the 10-bit wide INSTRUCTION PORT. The instruction word, I_{9-0}, is latched prior to the instruction decoder during phase one (clock HI) and is passed during phase two (clock LO). In addition to the microcode, two dedicated control pins affect the device's operation: the DSEL pin (see Y Port, D Port, and DSEL Control Pin); and the AIR Enable pin (see Alternate Instruction Register and AIR Enable). These pins are considered instruction bits, and latched as described above.

Y Port, D Port, and DSEL Control Pin

The ADSP-1410 has two 16-bit ports: a DATA (D) PORT and an ADDRESS (Y) PORT. The output drivers of both ports are three-state disabled unless an instruction specifies an output.

Addresses supplied to external data memory are output over the unidirectional Y port. The address supplied may come from one of three sources: an internal address (R) register, the data (D) port, or the ALU. The DSEL (Data Select) pin controls whether an R register (DSEL LO) or external data (DSEL HI) is the address source. The address source can either be directly output over the Y port, or passed through the ALU for modification prior to output (see Pre-Update Mode versus Post-Update Mode). Hardware three-state output control of the Y port is possible (see note in "Alternate Instruction Register and AIR Enable" section). Finally, the address being output (direct or modified source) may be bit-reversed (see Bit Reverser).

The Y port has two modes of operation (see Transparent Mode versus Latched Mode). In the more commonly used latched mode, addresses are latched during phase two (clock LO). The transparent mode disables the output latch and may be used in conjunction with stopping the clock LO, allowing data to be passed through (directly, or modified by the ALU) the AG without performing updates.

Any internal register may be read or written via the ADSP-1410's D port. Also, external data can be supplied to the chip over this port for immediate addressing purposes.

Note:
The ADSP-1410 may power-up driving the data bus. Caution should be used to avoid creating a bus contention with other devices which may be sharing this bus. To prevent bus contention, the CLK input may be forced LO during power-up (disabling the output data drivers). During this time, a RESET instruction should be setup at the instruction port to be executed as the first operation when the clock starts up.

Registers

The ADSP-1410 has 30 16-bit registers, organized into four banks. Single-cycle transfers between certain register banks are supported.

Sixteen ADDRESS (R) REGISTERS hold memory address pointers. In the same cycle that a 16-bit R value is output over the address (Y) port, it may be incremented, decremented, offset, modified by a logical operation, or left/right shifted by one bit. The updated value is then written back into the original R location (pre-update mode). In post-update mode, the address is output after being modified. Any R value (or data, using DSEL) may be bit-reversed on output.

Six OFFSET (B) REGISTERS furnish a second operand to the ALU (the other, provided by an R register or the data bus) for modifying the address to be output. The B registers are partitioned into two, user-selectable (see Control Register: B Bank Select) banks and external data can substitute as an offset value whenever B_3 (bank one) or B_7 (bank two) is used (see Table IV).

Four COMPARE (C) REGISTERS supply one source to the on-chip comparator, whose other source is the address being output. When an address moves to or beyond a boundary set by the C value, the CMP flag goes active (HI).

Four INITIALIZATION (I) REGISTERS can—conditional on the CMP flag going active—overwrite any R value, allowing overhead-free branches to the top of an addressing loop. Note that I and C registers are always paired. Conditional re-initializing of R registers may be independently inhibited for individual I registers (see Control Register CR_{3-0}).

ALU and Shifter

The ADSP-1410's 16-BIT ALU performs adds, subtracts, and logical operations. Usually, one source is an offset (B) register, while the other is an address (R) register. However, external data provided via the D port may substitute either for an R register (under the control of the DSEL pin), or a B register (using B3 or B7).

For two-chip/double-precision ALU operations, CARRIES into the MS chip and out of the LS chip (CS_{in} and CS_{out}) are conveyed via the Y_{15} pin (see Precision Modes).

The ALU also contains the logic required for single-bit SHIFTS of a supplied R register. Left shifts are logical, while right shifts are arithmetic. In two-chip/double-precision shift operations, the Y_{15} pin conveys the shifted bit. In single-precision operation, the carry/shift status of the device cannot be monitored.

The destination of an ALU or shift result is always the source R register location specified in the instruction—even if external data is the source. If the post-update mode is used, the ALU/shift result is sent directly over the address (Y) port on the current cycle (in addition to being returned to the source R location).

Alternate Instruction Register and AIR Enable

The ALTERNATE INSTRUCTION REGISTER (AIR) is a 10-bit register which may be loaded with any instruction. On any cycle that the AIR Enable pin is asserted, the device will execute the instruction held in the AIR, rather than the instruction set up on the instruction pins (except for the RST instruction).

The AIR's principal purpose is to conserve microcode. One way to conserve microcode is to load a frequently-used instruction (e.g., a looping instruction) into the AIR. Then, this instruction is executed simply by asserting the device's AIR Enable pin—temporarily suspending the need for external microcode.

The AIR can also conserve microcode in applications using multiple AGs (e.g., double-precision or high-throughput systems). If the AGs generally execute identical instructions, external microcode may be significantly reduced if they share a common microcode instruction field. During some cycles, however, it may be crucial for an AG to execute an instruction different from the common instruction—something which the AIR and its enable pin allow. For example, a NOP instruction can be loaded into an AG's AIR; anytime the AIR Enable pin is asserted, the AG will be selectively "put to sleep" (I/O pins three-state disabled; no change in internal state).

The AIR register may be read over the data port (D_{9-0}) in a single cycle. As Table I shows, the AIR may be written via the data port (D_{9-0}) or the instruction port. If the instruction written into the AIR is provided via the instruction port, two cycles are required. This method allows the AIRs of two or more AGs sharing microcode to be selectively loaded by differentially asserting their DSEL pins. Note that if the DSEL pin is LO during the *entire* second phase (clock LO) of the LDA instruction, no AIR loading occurs. This implicitly requires that DSEL be setup accordingly prior to the start of the LDA instruction, as it is latched during phase one (clock HI).

INSTRUCTION LOADED INTO THE AIR VIA THE:	
DATA PORT	**INSTRUCTION PORT**
1. Execute "Write AIR" instr.	1. Execute "Load AIR" instr. 2. Provide instr. on instr. port and assert DSEL pin.

Table I. Options for Reading and Writing the AIR

A second method exists for executing the instruction in the AIR. Looping instructions compare an address (R) value to a compare (C) value and, if the address has moved to or beyond a pre-set boundary, the CMP flag goes HI. If CR_{10} (see Control Register and Conditional AIR Execute Mode) is set, a true comparison causes the device to execute its next instruction from the AIR (see Table III.) This capability facilitates no-overhead modulo addressing (see application note: Modulo Addressing).

Note:
The AIRE pin may be used to control the Y port output drivers by loading a NOP into the AIR register; the AIRE pin becomes dedicated to three-state control of the Y port. This technique supports connection of multiple address sources to the same bus.

Flags and Comparator
The ADSP-1410 has two internal flags—CMP and ZERO—that share the external CMP/Z pin. The CMP flag, set by the comparator, is affected by looping instructions. The ZERO flag is set whenever a Logical Shift instruction has a zero result. In cycles that do not affect the CMP or ZERO flag, the CMP/Z flag pin defaults LO.

As Table II shows, the CMP flag goes HI whenever the supplied address moves to or beyond a boundary set by the specified C register. The address that is compared to the C value is always the address that is output—even in post-update mode. R, C, and B values are treated as unsigned integers by the Comparator.

Twos-Complement Offsets
Negative offsets are generally handled by the R⟵R − B instruction. However, if for some reason the user is interpreting offset values as negative twos complement numbers, the instruction R⟵R + B will cause the comparator to sense whether R≥C (when the condition R≤C is of interest). The user may account for this reversal (e.g., by monitoring for the CMP flag going LO, rather than HI), but looping instructions cannot be fully utilized.

ARITHMETIC OPERATION	CMP FLAG HIGH IF:
$R_n \leftarrow R_n + 1$ (YINC instruction)	$R_n \geq C_i$
$R_n \leftarrow R_n - 1$ (YDEC instruction)	$R_n \leq C_i$
$R_n \leftarrow R_n + B_m$ (YADD instruction)	$R_n \geq C_i$
$R_n \leftarrow R_n - B_m$ (YSUB instruction)	$R_n \leq C_i$

Table II. CMP Flag Truth Table

Alternating Offsets
If the microprogram switches between different offsets and the AG is in the normal, pre-update mode, the comparator logic may produce seemingly erroneous results because comparisons are not made until the cycle following the update. In pre-update mode, when a routine switches between positive and negative offsets, the comparator will check for wrong condition because the comparison is not made until the following cycle. The value in the compare register must anticipate the comparator sense reversal by one cycle.

Bit Reverser
Addresses can be bit-reversed as they are output, which is useful in algorithms such as the Fast Fourier Transform. The bit-reverse mapping is as follows, where K_i and Y_i denote the j^{th} bit of K (either an address register or the data bus) and Y (the address port), respectively.

$K_0 \rightarrow Y_{15}$
$K_1 \rightarrow Y_{14}$
•
•
•
$K_{15} \rightarrow Y_0$

Bit reversal only affects the value that appears on the address port; it does not affect the value returned to the R register location. The hardware bit reverser operates only on single-precision, 16-bit addresses. For details on software reversal of N-bit (N<16) fields, see the application note: Variable-Width Bit Reversing.

Control Register
The ADSP-1410's 11-bit CONTROL REGISTER (CR_{10-0}) may be read or written via the device's data port, D_{10-0}. Dedicated instructions are used to read or write the entire control register, or to set and clear individual bits (see Instruction Group 4). On power-up, the RST instruction clears the control register to all zeros automatically.

The following list shows the control register organization. If the bit(s) is set (HI), the specified mode is operative.

CR	Bit Assignment
3–0	*Re-Initialization Mask*: For looping instructions, enables conditional re-initialization of R registers with I registers. For example, setting bit 2 of the CR allows I_2 to re-initialize the selected R register if the address has moved to or beyond the boundary set by C_2.
5–4	*Precision Select*: 00 = single-precision mode, 01 = double-precision mode, LS chip; 10 = double-precision mode, MS chip; 11 = double-precision mode, single-chip.
6	*Transparent Mode*: Sets the address (Y) port to the transparent mode; otherwise, the Y port is latched during phase two.
7	*R Bank Select*: Selects the upper eight R registers as address sources for the YADD and YSUB instructions.
8	*B Bank Select*: Selects the upper four B registers as offset sources for all instructions.
9	*Post-Update Mode*: Sets the post-update mode (addresses supplied after updating).
10	*Conditional AIR Execute*: Sets the conditional AIR mode allowing looping instructions to (conditional upon the CMP status going true) be fetched from the AIR on the next instruction, rather than the instruction port. Using this mode disables conditional re-initialization (of R by I on CMP) and forces the default update of R.

ADSP-1410 OPERATING MODES

The flexibility of the ADSP-1410 is enhanced by several optional modes of operation. These modes, governed by the control register, are discussed in detail in this section.

Precision Modes

Typically, the ADSP-1410 provides single-precision (16-bit) addresses. If greater addressing range is needed, double-precision (30-bit) addresses can be supplied. Two double-precision modes are supported—one with two chips cascaded and the other with a single chip. Specific instructions set these modes. Double-precision (single- or two-chip) bit-reversing and is not supported.

Two-Chip Double-Precision

(CR_{5-4} = "01" for LS chip; "10" for MS chip). In this mode, two ADSP-1410's are cascaded to generate double-precision addresses at a rate of one per cycle. Each address may be output, incremented, decremented or modified by an offset value, compared to a double-precision value, and conditionally re-initialized by a double precision word. Alternately, double-precision logical/shift operations may be performed.

The Y and D ports of each chip are restricted to the lower 15 bits, freeing the MSBs of both devices to convey carry/shift and CMP status, respectively (see Figure 2). For double-precision adds/subtracts, the LS chip sends carry/borrow status over the Y_{15} pin; the MS chip uses Y_{15} to accept carry/borrow status from the LS. For left (right) shifts, the LS (MS) conveys the shifted bit over the Y_{15} pin.

Double-precision, conditional re-initializations are implemented by dedicating the D_{15} pin on each device's data port to receive the CMP status from the other. When performing a looping instruction, the MS chip generates a valid CMP flag on its CMP/Z output. For a logical or shift instruction, the CMP/Z outputs from both the LS and MS chips must be ANDed to produce a single valid ZERO flag. To ensure that this flag is valid on the next low-to-high transition of the clock, the output of the AND gate should be latched as shown in Figure 2. The ZERO flag is latched on the falling edge of the clock and held by the latch until the next falling edge.

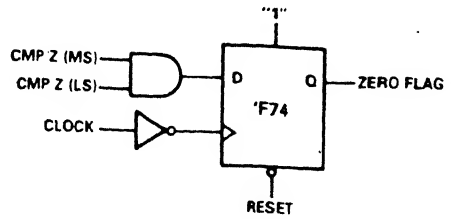


Figure 2. Valid Two-Chip Double-Precision ZERO (Logical Instructions)

In this mode, all values are 15-bit words. The 30-bit address is supplied in two 15-bit words over the Y_{14-0} pins of the two devices. Internally, the MS bit of each operand is zeroed prior to ALU operations, the MSB of the result then becoming the carry/shift bit. External data provided over the D port must be segmented with the 15 LSBs going to the LS chip and the 15 MSBs to the MS chip.

In two-chip/double-precision mode, both chips may share the same microcode instruction. The only complication to this sharing is in differentially initializing the MS and LS chips. Internal logic allows this initialization to be accomplished. Both chips are fed the instruction designating it as the LS chip. The assertion of DSEL on the intended MS chip during the SETP instruction reverses the two LS instruction bits (those defining the chip configuration to the control register), allowing both MS and LS designations to be performed simultaneously.

Single-Chip/Double-Precision

(CR_{5-4} = "11"). In this mode, double-precision (30-bit) addresses are generated at a rate of one every two cycles. Each address may be output, incremented or decremented, and compared to a double-precision compare (C) value. Logical/shift operations are also supported. Conditional re-initialization with I registers and the conditional AIR mode are not supported.

LSW operations are executed first, followed by MSW operations (with the exception of right shifts). Even-numbered R registers are reserved for LSWs, while odd registers are assumed to be MSWs. No such restrictions apply to B or C registers; MS *or* LS words may be held in *any* B or C register, but such allocation must be tracked by the user. After an operation involving LSW registers, the device stores the carry/shift bit (as appropriate) needed to complete the double-precision operation. On the next operation involving MSW registers, this intermediate value is utilized. Storage of the carry/shift bit occurs only on LSW operations, except for double-precision right shifting, which starts with the MSW. If non-addressing operations intervene, the intermediate value is not disturbed. The comparator will generate a meaningful CMP signal after each MSW operation.

In this mode, only the 15 LSBs of any register are used. The LSW and MSW addresses that are supplied are both 15-bit words. The Y_{15} (MSB) pin of the 16-bit address port designates

whether the address is the LSW (= 0) or MSW (= 1), and may be used to control an external mux. Note that the MSB of values provided via the data (D) port is not meaningful in this mode.

Transparent Mode
(CR_6 HI). In this mode, the address port is made transparent during the entire cycle, rather than only phase one. The transparent mode may also be used in conjunction with stopping the clock (I.O.), in which case the entire device behaves asynchronously and no updates are written internally.

Latched Mode
(CR_6 LO). In latched mode, output values are enabled during phase one and latched at the address (Y) port during phase two.

Use of the latched mode guarantees that outputs remain stable throughout the current cycle regardless of changes at the instruction port. This, in contrast to the transparent mode, in which such changes may occur quickly enough to alter the output before cycle end.

Post-Update Mode
(CR_9 HI). Addresses are output after the update operation. The delay between the start of phase one and output of a valid address is extended in this mode to allow for updating. The addresses output are equivalent to the values written back into the specified address (R) register. In this mode, external data may be brought on chip, modified and output—in a single clock cycle.

Pre-Update Mode
(CR_9 LO). This is the normal update mode in which addresses are output over the address (Y) port prior to update operations (increment, decrement, offset, shift, and logical)—allowing addresses to be generated at maximal speed. Note however, that this mode requires two cycles to bring external data on chip, modify it, and supply it as an address.

Conditional AIR Execute Mode
(CR_{10} HI). In this mode, a valid CMP flag on looping instructions causes the next instruction to be executed from the AIR. The MODULO ADDRESSING section highlights a particularly valuable use of this mode.

Note that conditional re-initialization of address registers is disabled when using the conditional AIR execute mode. The default (ELSE clause) is performed unconditionally whether or not the instruction is from the instruction port or the AIR).

(CR_{10} LO). Conditional AIR execution is disabled. Conditional re-initialization is fully operational, contingent upon the re-initialization mask (CR_{3-0}).

Table III summarizes the different ways the CMP status affects operation of the AG as a function of the conditional AIR execute mode control bit, CR_{10}, and the re-initialization mask, CR_{3-0}.

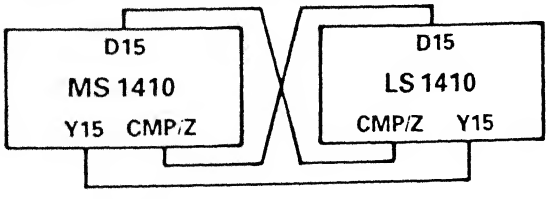


Figure 3. Two-Chip Double Precision Handshaking

CMP STATUS	CR_{10} LO		CR_{10} HI
	CR_j LO	CR_j HI	
LO	No Effect	No Effect	No Effect
HI	CMP/Z goes HI	CMP/Z goes HI; $R_n \leftarrow I_j$	CMP/Z goes HI; Next instr. executed from AIR

Table III. Effect of Compare (CMP) Status for Looping Instructions; Note: j = 3–0, the Re-Initialization Mask.

INSTRUCTION SET DESCRIPTION

The ADSP-1410's instruction set is partitioned into six groups, which are discussed below. First, however, issues spanning several instruction groups are discussed.

Most of the instruction groups contain instructions using one of the chip's six offset (B) registers. Without exception, these instructions have just two bits available for selecting the B register. Consequently, offset registers are partitioned into two banks. The upper/lower bank selection is maintained in the control register (CR_8) and is set or cleared by dedicated instructions. Whenever the "fourth" B register of either bank is specified (B_3 or B_7), the ALU's offset source becomes external data (see Table IV).

CR_8 & TWO-BIT OFFSET (B) REGISTER FIELD	OFFSET SOURCE
0 00	B0
0 01	B1
0 10	B2
x 11	Data Port*
1 00	B4
1 01	B5
1 10	B6
x 11	Data Port*

Table IV. Offset Value Structure

*Explicit use of DSEL is unnecessary when using B_3 or B_7 offsets; the offset data is sourced from the data bus by default.

In several instruction groups (see mnemonics and opcodes for details), address (R) registers are used. In all cases, asserting the DSEL pin allows external data to be substituted for an R value as both output and update data.

Two instruction groups (looping and logical/shift) both supply and update the address. Normally, addresses are supplied prior to updating (pre-update). In post-update mode however, the addresses are output after the update operation is performed. CR_0 controls this mode of operation.

For all instructions accessing an offset register, the MS bit of the three-bit offset register address (B, of Bbb) is fetched from the control register and is programmed by the SELB instruction. This is also the case for the YADD and YSUB instructions (group 1) as pertains the MS bit of the four-bit address register address (R, of Rrrr), programmed by the SELR instruction. In both cases, it is incumbent upon the programmer to ensure the appropriate register bank is selected.

The Y port is only driven on output instructions (mnemonic form Yxxx, see MNEMONICS AND OPCODES). Otherwise, the Y port defaults to a high-impedance state.

Instruction Group 1: Looping

Instructions in the looping group supply the contents of a selected address (R) register to the address (Y) port and then overwrite the R location with an updated value.

All instructions in this group generate an internal CMP status indicating whether the supplied address has moved to or beyond the boundary specified by the compare register. This status may be monitored externally via the CMP/Z pin. Internal to the chip, the CMP status can i) be ignored, ii) be used to control re-initialization of the R register value with a selected I register value (e.g., to restart an addressing loop), or iii) control execution of an instruction located in the AIR on the next cycle. Individual control register bits determine which option is enforced (see Control Register).

YINC Output & Increment/Init.

Pre-Update Mode: $Y \leftarrow R_n$;
IF ($R_n \geq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n + 1$.

Post-Update Mode: $Y \leftarrow R_n + 1$;
IF ($Y \geq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n + 1$.

Output an address (R) register on the address (Y) port and compare it to one of the compare (C) registers. If the address is less than C_j, the R location is simply updated with an incremented value. However, if $R_n \geq C_j$, CMP status goes HI and the R register is re-initialized with the I_j value, provided the initialization mask (CR_{3-0}) is enabled for I_j. Note that other modes of operation allow CMP status to be ignored (e.g., the instruction executed is simply "$Y \leftarrow R_n$; $R_n \leftarrow R_n + 1$") or to cause the AIR instruction to execute on the next cycle.

YDEC Output & Decrement/Init.

Pre-Update Mode: $Y \leftarrow R_n$;
IF ($R_n \leq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n - 1$.

Post-Update Mode: $Y \leftarrow R_n - 1$;
IF ($Y \leq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n - 1$.

Same as above except the R value is decremented instead of incremented; CMP is valid if the R value is less than or equal to the C value.

YADD Output & Add Offset/Init.

Pre-Update Mode: $Y \leftarrow R_n$;
IF ($R_n \geq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n + B_m$.

Post-Update Mode: $Y \leftarrow R_n + B_m$;
IF ($Y \geq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n + B_m$.

Same as YINC except the R value is summed with the contents of a selected offset (B) register.

The R register bank select bit (CR_7) is used in both the YADD and YSUB (offset) instructions.

YSUB Output & Subtract Offset/Init.

Pre-Update Mode: $Y \leftarrow R_n$;
IF ($R_n \leq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n - B_m$.

Post-Update Mode: $Y \leftarrow R_n - B_m$;
IF ($Y \leq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n - B_m$.

Same as YADD except the selected offset (B) register is subtracted from the R value.

Instruction Group 2: Register Transfers

Instructions in the register transfer group support internal register transfers, as well as transfers between internal and external registers. Internally, any I or B register may be written directly to any R register. Also, any R register may simultaneously be output and written directly to a B or C register. For an R-to-R transfer, the source R register can first be written to a B register, followed by a write of the B register to an R register on the next cycle.

Internal registers are read or written externally via the bi-directional data port. There are explicit instructions to read any of these registers; however, only the I registers have an explicit Write instruction. The R, B, and C registers may be written with external data by executing a transfer instruction (YRTR, YRTB, and YRTC) and asserting the DSEL pin, substituting the external data for the designated R value.

YRTR Output & Transfer Addr. Reg. to Self

$Y \leftarrow R_n$

Outputs selected address (R) register over the address (Y) port. When DSEL is asserted, data port values are output and, in the same cycle, written into the selected R register.

YRTB Output & Transfer Addr. Reg. to Base Reg.

$Y \leftarrow R_n$; $B_m \leftarrow R_n$

Outputs selected R register over the Y port and copies it into a selected B register. When DSEL is asserted, data port values are output and, in the same cycle, written into the selected B register.

YRTC Output & Transfer Addr. Reg. to Comp. Reg.

$Y \leftarrow R_n$; $C_j \leftarrow R_n$

Same as above, except that values are written to a C register.

DTI Transfer Data Bus to Init. Reg.
$I_i \leftarrow D$

Loads selected I register from data (D) port.

ITR Transfer Init. Reg. to Addr. Reg.
$R_n \leftarrow I_i$

Selected R register is loaded from an I register, allowing a microprogram to restart a loop at any time.

BTR Transfer Base Reg. to Addr. Reg.
$R_n \leftarrow B_m$

Loads an R register from a B register. Once in the R register, the B value may be modified and then returned to the B file (using a YRTB instruction). Recall, use of B_3 or B_7 will access the data port as the offset source, allowing R registers to be initialized directly from the data port.

RTD Transfer Addr. Reg. to Data Bus
$D \leftarrow R_n$

Supplies selected R register to data (D) port.

CTD Transfer Comp. Reg. to Data Bus
$D \leftarrow C_i$

Supplies selected C register to data (D) port.

BTD Transfer Base Reg. to Data Bus
$D \leftarrow B_m$

Supplies selected B register to data (D) port.

ITD Transfer Init. Reg. to Data Bus
$D \leftarrow I_i$

Supplies selected I register to data (D) port.

Instruction Group 3: Logical & Shift
Instructions in the logical shift group supply a value from a selected address (R) register to the address (Y) port and then unconditionally overwrite the selected R location with a modified version of the output. Modify operations include logical (AND, OR, and XOR) and shift (one-bit left/right) operations. All instructions in this group affect the ZERO flag, which goes HI if the result of the modification is zero. The ZERO flag status is available externally over the CMP/Z pin.

YOR Output & Logical OR to Addr. Reg.
$Y \leftarrow R_n; R_n \leftarrow (R_n \text{ OR } B_m)$

Selected R register is supplied to the address (Y) port; the specified R location is then overwritten with the logical OR of the B register and original R value.

YAND Output & Logical AND to Addr. Reg.
$Y \leftarrow R_n; R_n \leftarrow (R_n \text{ AND } B_m)$

Same as above, except that a logical AND is performed.

YXOR Output & Logical XOR to Addr. Reg.
$Y \leftarrow R_n; R_n \leftarrow (R_n \text{ XOR } B_m)$

Same as above, except that a logical XOR is performed.

YASR Output & Arithmetic Right Shift to Addr. Reg.
$Y \leftarrow R_n; R_n \leftarrow \text{ASR}(R_n)$

Selected R register is supplied to the address (Y) port; the specified R location is then overwritten with the original R value arithmetically shifted right (ASR) by one bit (the MSB is repeated).

YLSL Output & Logical Left Shift to Addr. Reg.
$Y \leftarrow R_n; R_n \leftarrow \text{LSL}(R_n)$

Selected R register is supplied to the address (Y) port; the specified R location is then overwritten with the original R value logically shifted left (LSL) by one bit (the LSB is zero-filled).

Instruction Group 4: Control Register
Instructions in the control register group reset, read, and write the entire control register or individual control register bits (see Control Register).

Note the use of "x" and "pp" to denote values supplied within the opcode field (see MNEMONICS AND OPCODES). A positive logic convention is used throughout.

RST Reset Control Reg.
$CR \leftarrow 0$

Clears the entire control register (CR_{10-0}). The RST instruction has dedicated decoding logic so that it takes precedence even over the second instruction of a conditional AIR sequence.

DTCR Transfer Data Bus to Control Reg.
$CR \leftarrow D$

Writes the entire control register (CR_{10-0}) from the data port, D_{10-0}.

CRTD Transfer Control Reg. to Data Bus
$D \leftarrow CR$

Outputs the entire control register (CR_{10-0}) over the data port, D_{10-0}.

SETI Set/Clear Conditional Init. on CMP Flag
$CR_{ji} \leftarrow x$

Enables conditional re-initialization of an R location, subject to CMP status (see Control Register). This instruction loads the x value into the control register bit specified by jj. Conditional re-initialization of address registers by the C_{jj}/I_{jj} pair is inhibited if the corresponding CR_{jj} is cleared.

SETP Set Chip precision
$CR_{5-4} \leftarrow pp$

Loads a 2-bit code (pp) into control register bits 5 and 4, specifying the addressing mode of the device:

00 = single-precision mode;
01 = double-precision mode, LS chip (10 if DSEL);
10 = double-precision mode, MS chip;
11 = double-precision mode, single-chip.

If the instruction "SETP, 01" is supplied and the MS chip's DSEL pin is asserted, the CR_{5-4} bits are reversed, i.e., the MS chip is loaded with "10", not "01" (see Precision Modes). This is useful if the MS and LS chips share a common instruction bus.

SETY Set Y Port to Transparent/Latched Mode
$CR_6 \leftarrow x$

Uses the LS instruction bit to set the address (Y) port to the transparent (HI) or latched (LO) mode. This status is maintained in control register bit 6.

SELR Select Upper/Lower Addr. Reg. Bank
$CR_7 \leftarrow x$

The LS bit of this instruction provides the missing Address (R) register select bit required by the YADD and YSUB instructions. This selection is maintained in control register bit 7.

SELB Select Upper/Lower Base Reg. Bank
$CR_8 \leftarrow x$

The LS bit of this instruction provides the missing B register select bit required by all instructions utilizing offset (B) registers. This selection is maintained in control register bit 8.

SETU Set Update Mode (Post/Pre)
$CR_9 \leftarrow x$

Setting this bit causes the chip to output address values after updating them (post-update mode). The LS bit of this instruction determines the value of control register bit 9.

SETA Set/Clear Conditional AIR Execute Mode
$CR_{10} \leftarrow x$

Setting this bit causes Looping instructions—conditional on CMP status being HI—to execute the following instruction from the AIR on the next cycle. In this mode, conditional re-initialization of R by I on CMP is inhibited. The LS bit of this instruction determines the value of control register bit 10.

Instruction Group 5: AIR Control

Instructions in the AIR group write and read the Alternate Instruction Register (AIR). The AIR may be written or read over the data bus in one cycle or written via the instruction port in two cycles (see Table I). The instruction contained in the AIR is executed whenever the AIR Enable pin is asserted or on the next cycle in the conditional AIR execute mode.

WRA Write AIR with Data Bus
$AIR \leftarrow D$

Write the AIR from the data (D) bus (D_{9-0}).

RDA Read AIR at Data Bus
$D \leftarrow AIR$

Read the AIR over the data (D) bus (D_{9-0}).

LDA Load AIR from Instruction Port on Next Cycle (Requires DSEL HI)
AIR ← Instruction Port

This instruction is the first of a two-cycle sequence that loads the AIR via the instruction port. On the cycle following the execution of LDA, the instruction at the instruction port is loaded into the AIR (and not executed). DSEL must be asserted with the LDA instruction (meeting the same setup and hold time requirements); otherwise, the AIR is not loaded. In systems with multiple ADSP-1410s sharing microcode instructions, this feature allows you to select particular devices for AIR loading.

Instruction Group 6: Miscellaneous

YDTY Pass Data Bus to Y Port
$Y \leftarrow D$

Data (D) port values are supplied directly to the address (Y) port. Note that internal address (R) registers are not affected by this instruction.

YREV Output Addr. Reg. in Bit-Reversed Format
$Y \leftarrow YREV(R_n)$; $R_n \leftarrow R_n + B_m$

The selected address (R) register is bit reversed at the output port. The original (unreversed) R value is added to the selected offset (B) register, and written back into the specified R location. Condition testing is not performed. Bit reversing affects only output data, not register contents.

NOP No Operation

Prevents any changes to the internal conditions of the AG. All I/O pins go to the three-state disable mode.

ADDRESS GENERATOR APPLICATIONS

The ADSP-1410 has a wide range of uses in high-speed digital signal processing and general purpose computer applications. In particular, this AG can be used in implementing the following:

Circular Data Buffers
- FIR filter tapped delay lines
- Correlator delay lines
- Image processing delay lines
- Recirculated data I/O for transient data capture or stimulus source

Memory Management
- Fast Fourier Transform data and twiddle factors
- Matrix computations

Table Look-Ups

Masking and table address mapping with AND/OR and bit reverse capabilities.

Variable-Width Bit Reversing

The internal bit-reversing multiplexer of the AG accommodates only full, 16-bit addresses (64K FFTs). For smaller FFTs, (utilizing a right-justified subset of the 16-bit address field), a zero-overhead software approach may be employed. The details of this approach may be found in the application note: "Variable-Width Bit Reversing with the ADSP-1410 Address Generator". Essentially, the technique is this: an R register is intialized with the bit-reversed value of the 16-bit starting address (a "pre-reversed" version of the first data point location) and a B register with the value $K \cdot 2^{16-N}$, where K is the step size between samples and N is the order of the FFT. Now, repeated execution of the YREV instruction will output the appropriate bit-reversed addresses; updating the R register each time.

Multi-Tasking Operations

Context switching allowed by large number of on-chip registers or by instructions allowing all registers to be saved and restored.

16-Bit ALU/Accumulator

By substituting external data for a B register and operating in post-update mode, ALU operations can be performed at high speed. ALU sources are the external data and any one of sixteen internal R registers. Results are stored on-chip in these R registers. Two chips may be cascaded for double-precision operations.

ANALOG DEVICES

64-Bit IEEE Floating-Point Chipsets

ADSP-3210/3211/3220/3221

FEATURES

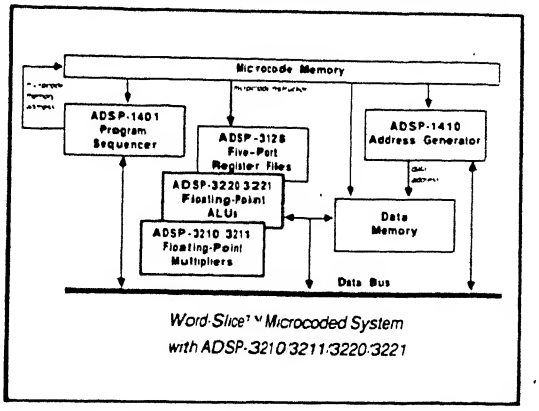
- Complete Chipsets Implementing Floating-Point Arithmetic: Two Multiplier Options and Two ALU Options
- Fully Compatible with IEEE Standard 754
- Arithmetic Operations on Four Data Formats:
  - 32-Bit Single-Precision Floating-Point
  - 64-Bit Double-Precision Floating-Point
  - 32-Bit Twos-Complement Fixed-Point
  - 32-Bit Unsigned Fixed-Point
- Only One Internal Pipeline Stage
- High-Speed Pipelined Throughput
  - Single-Precision and Fixed-Point Multiplication Rates to 20 MFLOPS
  - Double-Precision Multiplication Rates to 5 MFLOPS
  - Single-, Double-, and Fixed-Point ALU Rates to 10 MFLOPS
- Low Latency for Scalar Operations
  - 140ns for 32-Bit Multiplier Operations
  - 315ns for 64-Bit Multiplier Operations
  - 240ns for 32-Bit ALU Operations
  - 290ns for 64-Bit ALU Operations
- IEEE Divide and Square Root (ADSP-3221 ALU)
- Flexible I/O Structures:
  - ADSP-3211/3220/3221: Either One or Two Input-Port Configuration Modes
  - ADSP-3210: One Input Port
- 750mW Max Power Dissipation per Chip with 1.5μm CMOS Technology
- 100-Lead Pin Grid Array (ADSP-3210 Multiplier)
- 144-Lead Pin Grid Array (ADSP-3211/3220/3221)
- Available Specified to MIL-STD-883, Class B

APPLICATIONS

- High-Performance Digital Signal Processing
- Engineering Workstations
- Floating-Point Accelerators
- Array Processors
- Mini-supercomputers
- RISC Processors

GENERAL DESCRIPTION

The ADSP-3210/3211 Floating-Point Multipliers and the ADSP-3220/3221 Floating-Point ALUs are high-speed, low-power arithmetic processors conforming to IEEE Standard 754. A chipset consisting of either Multiplier used with either ALU contains the basic computational elements for implementing a high-speed numeric processor. Operations are supported on four data formats: 32-bit IEEE single-precision floating-point, 64-bit IEEE double-precision floating-point, 32-bit twos-complement fixed-point, and 32-bit unsigned-magnitude fixed-point.



Word-Slice™ Microcoded System with ADSP-3210/3211/3220/3221

The high throughput of these CMOS chips is achieved with only a single level of internal pipelining, greatly simplifying program development. Theoretical MFLOPS rates are much easier to approach in actual systems with this chip architecture than with alternative, more heavily pipelined chipsets. Also, the minimal internal pipelining in the ADSP-3210/3211/3220/3221 results in very low latency, important in scalar processing and in algorithms with data dependencies. To further reduce latency, input registers can be read into the chips internal computational circuits at the rising edge that loads them from the input port (formerly called direct operand feed).

In conforming to IEEE Standard 754, these chips assure complete software portability for computational algorithms adhering to the Standard. All four rounding modes are supported for all floating-point data formats and conversions. Five IEEE exception conditions – overflow, underflow, invalid operation, inexact result, and division by zero – are available externally on status pins. The IEEE gradual underflow provisions are also supported, with special instructions for handling denormals. Alternatively, each chip offers a FAST mode which sets results less than the smallest IEEE normalized values to zero, thereby eliminating underflow exception handling when full conformance to the Standard is not essential.

The instruction sets of the ADSP-3210/3211/3220/3221 are oriented to system-level implementations of function calculations. Specific instructions are included to facilitate such operations as floating-point division and square root, table lookup, quadrant normalization for trig functions, extended-precision integer operations, logical operations, and conversions between all data formats.

The ADSP-3210 Floating-Point Multiplier is a one input- and one output-port device with four input registers. The ADSP-3211 Floating-Point Multiplier adds a second input port and doubles the number of input registers to eight. It executes all ADSP-3210

Word-Slice is a trademark of Analog Devices, Inc.

STATUS FLAGS

The ADSP-3210/3211/3220/3221 chipset generates on dedicated pins the following exception flags specified in the IEEE Standard: Overflow (OVRFLO), Underflow (UNDFLO), Inexact Result (INEXO), and Invalid Operation (INVALOP). The IEEE exception condition Division-by-Zero is flagged by the simultaneous assertion of both OVRFLO and INVALOP pins. The five IEEE exceptions are defined in accordance to the default assumption of Std 754 of nontrapping exceptions.

These four flag results are registered in the Status Output Register when the results they reflect are clocked to the Output Register. They are held valid until the next rising clock edge. The IEEE Standard specifies that exception flags when set remain set until reset by the user. For full conformance to the standard, the status outputs from this chipset should be individually latched externally.

Denormal Input

In addition to the IEEE status flags, the ADSP-3210/3211 Multipliers have a DENORM output flag that signals the presence of a denormalized number at one of the input registers being read into the multiplier array. This denormal must be wrapped by the ALU before the Multiplier can read it. To minimize the system response time to a denormal input exception, the DENORM flag comes out earlier than the associated IEEE status flags. DENORM is normally in an indeterminate state. For single-precision multiplications, DENORM goes HI during the cycle after a denormal was read into the array (with the RDA/B controls). See Figure T4. For double-precision multiplications, DENORM goes HI during the third cycle after a denormal was read into the multiplier array. See Figure T5. Both Multipliers produce ZERO results under these conditions. The DENORM flag is asserted in both IEEE and FAST modes.

Some multiplications with denormal operands do not require wrapping and therefore do not cause the assertion of the DENORM flag. These are DNRM•ZERO, DNRM•INF, and DNRM•NAN. Multiplication of a finite number by zero always yields zero – the result the Multiplier will produce anyway – so there is no need to signal an exception. Any finite number multiplied by INF should yield INF, and the ADSP-3210/3211 Multipliers will produce this result with a DNRM operand, hence no wrapping is required. And multiplication of any number by a NAN produces a NAN (and the INVALOP flag); no wrapping is necessary for the Multipliers to produce this correct IEEE result.

Note that the ALUs in general operate directly on denormals and therefore do not flag any exception. The ADSP-3221 ALU, however, cannot operate directly on denormals in its division and square root operations. For these operations, denormal inputs will cause the simultaneous assertion of UNDFLO and INVALOP in IEEE mode. For divisions, INEXO HI indicates that the dividend is a DNRM; INEXO LO indicates that the divisor or both operands are DNRMs. In FAST mode, only INVALOP will be asserted. This denormal exception information becomes available with the status outputs, i.e., at the end of an attempted multicycle division or square root. In both modes for both division and square root, a properly signed all-ones NAN will be produced.

Invalid Operation and NAN results

INVALOP is generated whenever attempting to execute an invalid operation, as defined in Std 754 Section 7.1. The INVALOP output is also used in conjunction with other pins to indicate the Division-by-Zero exception and denormal divisor or dividend. The default nontrapping result is required to be a quiet NAN. Except when passing a NAN with PASS or copying a sign bit to a NAN, the ADSP-3210/3211/3220/3221 chipset will always produce a NAN with an exponent and fraction of all ones as a result of an invalid operation.

Conditions that cause the assertion of INVALOP are:

- NAN input read to computational circuitry (except for logical PASS)
- Multiplication of either ± INF by either ± ZERO
- In FAST mode, multiplication of either ± INF by either ± DNRM
- Subtraction of liked-signed INFs or addition of opposite-signed INFs
- Conversion of a NAN or INF to fixed-point
- Wrapping an operand that is neither a denormal nor ZERO
- Division of either ± ZERO by either ± ZERO or of either ± INF by either ± INF
- Attempting the square root of a negative number
- In conjunction with OVRFLO, the Division-by-Zero exception
- In FAST mode, a denormal divisor or dividend. In IEEE mode, in conjunction with UNDFLO, a denormal divisor or dividend
- In conjunction with UNDFLO, a denormal input operand to square root

Division-by-Zero

The Division-by-Zero exception is generated whenever attempting to divide a finite non-zero dividend by a divisor of zero (Std 754 Section 7.2). The Division-by-Zero exception is indicated on the ADSP-3221 ALU by the simultaneous assertion of both OVRFLO and INVALOP. The ALU result is always a correctly signed INF.

Overflow

OVRFLO is generated whenever the unbounded (i.e., supposing hypothetically no bounds on the exponent range of the result), post-rounded result exceeds in magnitude NORM.MAX in the destination format, as defined in Std 754 Section 7.3. Note that the overflow condition can occur both during computations and during data format conversions. The result will be either ± INF or ± NORM.MAX, depending on the sign of the result and the operative rounding mode. (See "Rounding – RND Controls" above.) The OVRFLO pin is also used to signal additional exception conditions.

Conditions that cause the assertion of OVRFLO are:

- Unbounded, post-rounded result exceeds destination format in computation or conversion
- In conjunction with INVALOP, the Division-by-Zero exception on the ADSP-3221 ALU
- Comparison when operand A is greater than operand B
- Exponent subtraction when the resultant exponent is more positive than can be represented in the destination format
- Twos-complement fixed-point additions and subtractions that overflow

Note that OVRFLO is always LO when the ADSP-3210/3211 Multipliers are in fixed-point mode.

Underflow

Underflow is defined in four ways in Std 754 Section 7.4. The IEEE Standard allows the implementer to chose which definition of underflow to use and provides no guidance. The first option is whether to flag underflow based on results before or after rounding. Consistent with the definition of overflow, underflow is always flagged with this chipset based on results *after* rounding (except for the operations of conversion from floating-point to fixed-point and logical downshifts). Thus, a result whose infinitely precise value is less than NORM.MIN yet which rounds to NORM.MIN will *not* be considered to have underflowed.

The second option is how to interpret what the Standard calls an "extraordinary loss of accuracy." The first way is in terms of the creation of non-zero, post-rounded numbers smaller in magnitude than NORM.MIN. The second way is in terms of loss of accuracy when representing numbers as denormals. With the ADSP-3210/3211/3220/3221 chipset, the conditions under which UNDFLO is asserted depend on whether the chip in question can generate denormals in its current operating mode. If the chip cannot generate denormals, the definition in terms of numbers smaller in magnitude than NORM.MIN will apply; if it can generate denormals, the definition in terms of inexact denormals will apply. Thus, which definition applies will depend on whether the chipset is operating in IEEE or FAST mode, whether its result is generated by a Multiplier or an ALU, and whether the operation is division.

With the ADSP-3210/3211 Multipliers, UNDFLO is generated whenever the unbounded, post-rounded, non-zero result is of lesser magnitude than NORM.MIN in the destination format, both in FAST and IEEE modes. In FAST mode, the data result will be ZERO; in IEEE mode the data result will be in the wrapped format. An exact ZERO result will never cause the assertion of UNDFLO.

With the ADSP-3220/3221 ALUs in the FAST mode, UNDFLO is also generated whenever the unbounded, post-rounded, non-zero result is of lesser magnitude than NORM.MIN in the destination format for standard ALU operations as well as for division and square root. For FAST mode underflows, the ALU result will always be ZERO.

With the ADSP-3220/3221 ALUs in IEEE mode, UNDFLO is generated (except for divisions) whenever the unbounded, infinitely precise (i.e., supposing hypothetically no bounds on the precision of the result), post-rounded result is a denormal and does not fit into the denormal destination format *without a loss of accuracy*. In other words, UNDFLO will be generated whenever an inexact denormal result is produced. (See "Inexact" below.) If the result is a denormal and does fit exactly, neither UNDFLO nor INEXO will be asserted. Note that additions, subtractions, and comparisons cannot generate this underflow condition (since no operand contains significant bits of lesser magnitude than DNRM.MIN). IEEE-mode ALU underflow exceptions occur only during conversions and divisions.

The division operation is treated like a multiplication operation in IEEE mode rather than an ALU operation in the definition of underflow. A quotient from division smaller in magnitude than NORM.MIN will always be flagged as underflowed with the ADSP-3221 ALU. The data result will be in the wrapped format. Note that $\sqrt{(DNRM.MIN)} \geq NORM.MIN$. Therefore, square root will never underflow with operands greater than or equal to DNRM.MIN.

Conditions that cause the assertion of UNDFLO are:

- With the ADSP-3210/3211 Multipliers, whenever the unbounded, post-rounded, non-zero result is of lesser magnitude than NORM.MIN in the destination format
- With the ADSP-3220/3221 ALUs in the FAST mode, whenever the unbounded, post-rounded, non-zero result is of lesser magnitude than NORM.MIN in the destination format
- With the ADSP-3220/3221 ALUs in IEEE mode, whenever an inexact denormal is produced or whenever the unbounded, post-rounded, non-zero quotient from division is of lesser magnitude than NORM.MIN in the destination format
- Conversions to integer if the magnitude of the floating-point source *before* rounding is less than one
- Conversions from DP floating-point to SP floating-point whenever the unbounded, post-rounded, non-zero result is less than SP DNRM.MIN or whenever an inexact denormal is produced.
- Comparison when operand A is less than operand B
- Attempting to wrap a ZERO
- Unwrapping if there is a loss of accuracy
- Exponent subtraction when the resultant exponent is more negative than can be represented in the destination format
- Logical downshift that before rounding would have shifted all bits out of the destination format
- In conjunction with INVALOP, a denormal divisor or dividend
- A quotient from division less than NORM.MIN
- In IEEE mode, in conjunction with INVALOP, a denormal input operand for square root

Inexact

The inexact exception is defined in Std 754 Section 7.5 as the loss of accuracy of the unbounded, infinitely precise result when fitted to the destination format. It is signalled on the ADSP-3210/3211/3220/3221 chipset by INEXO.

For fixed-point operations, the ADSP-3210/3211 Multipliers will assert INEXO HI if and only if any of the least-significant 32 bits of prerounded 64-bit products are ones. They never assert INEXO for logical operations. The ADSP-3220/3221 ALUs never assert INEXO for fixed-point or logical operations.

In an ADSP-3221 division operation, either a denormal divisor or a denormal dividend will cause the simultaneous assertion of UNDFLO and INVALOP. INEXO will, in that context, signal which of the two was the denormal: INEXO LO indicates that the divisor is a denormal; INEXO HI indicates that the dividend is a denormal.

Conditions that cause the assertion of INEXO are:

- Loss of accuracy when fitting result to destination format
- For fixed-point operations, the prerounded multiplier 64-bit product contains ones in the least-significant 32 bits
- In IEEE mode, in conjunction with both UNDFLO and INVALOP, dividend is a denormal (HI) or divisor is a denormal or both are denormals (LO)

Less Than, Equal, Greater Than, and Unordered

For comparison operations in the ALUs, the OVRFLO, UNDFLO, and INVALOP status outputs are used to indicate the four comparison conditions of IEEE Std 754, Section 5.7. They are defined as follows:

- "Less than" is signalled by the assertion of UNDFLO (while OVRFLO is LO)
- "Equal" is signalled by *not* asserting either OVRFLO or UNDFLO (i.e., both LO)
- "Greater than" is signalled by the assertion of OVRFLO (while UNDFLO is LO)
- "Unordered" is signalled by the assertion of INVALOP, caused by attempting a comparison with at least one NAN operand

The data result from a comparison operation is identical to subtracting operand B from operand A. See Tables X1 and X11.

In IEEE comparisons, the data types are always ordered in ascending sequence: −INF, −NORM, −DRNM, ZERO, DNRM, NORM, and INF. Comparisons between like signed INFs will generate the "Equal" status condition. Comparisons between signed ZEROs will also generate the "Equal" status. Any comparison to a NAN will also cause INVALOP and produce an all-ones NAN. Even in FAST mode, DNRMs will be compared based on their true value (rather than all being treated as ZEROs).

Special Flags for Unwrapping

The ADSP-3210/3211 generates a Round Carry Propagation Out flag, RNDCARO, that indicates whether or not a carry bit propagated into the destination format's fraction during the Multiplier's floating-point rounding operation. The rounding that the Multiplier does in creating the wrapped or unnormal result may cause a carry bit into the LSB in the destinations format's fraction. This rounding position will not in general be correct for a properly rounded denormal. Thus, when the underflowed Multiplier result is unwrapped to a denormal, the ALU has to undo the Multiplier's rounding and re-round to achieve the properly rounded denormal.

To do this, the ALU has to know if any carry bits in the Multipliers rounding operation propagated into the fraction of the result. This information is provided in the Multiplier's RNDCARO flag. The ALU also needs to know if the Multiplier's rounded result caused a loss of accuracy when expressed in its destination wrapped format, indicated by the Multipliers Inexact Result (INEXO) flag.

The ADSP-3220 3221 ALUs have a corresponding pair of flag status input pins: Round Carry Propagation In (RNDCARI) and Inexact Data In (INEXIN). In an unwrap operation, these flags are used by the ALU when converting from a WNRM to a DNRM to obtain the properly rounded result. RNDCARI and INEXIN should be setup to the ALU with the instruction for the unwrap operation. Both Multiplier and ALU must be using the same rounding mode.

The ADSP-3221 ALU itself generates WNRMs in underflowed division operations. These WNRMs must be fed back to the ALU to be unwrapped to DNRMs. The ADSP-3221, unlike the Multipliers, does not have a RNDCARO pin to signal whether or not a carry bit propagated into the destination format on rounding. For this reason, WNRMs produced by the ADSP-3221 ALU in division are rounded differently than they are on the Multipliers; underflowed (only) quotients are always truncated (Round-toward-Zero) to the destination wrapped format. Hence there is no carry bit propagation. When unwrapping a WNRM produced in division, RNDCARI should always be held LO. INEXIN should reflect the status of INEXO when the ALU produced the underflowed wrapped quotient.

The ADSP-3221 ALU also uses the RNDCARI and INEXIN pins to indicated wrapped A and B operands, respectively, to division and square root operations. Both RNDCARI and INEXIN should be held LO except for unwrap, division, and square root operations.

INSTRUCTIONS AND OPERATIONS

The ADSP-3210/3211 Multipliers execute the same instruction every cycle: multiply. It need not be specified explicitly in microcode. The data format of results and status flags from multiplication are shown in Tables IX and X. Note that double-precision floating-point multiplications are multicycle operations. Data must be available in the input registers as shown above in Figure 23.

Denormal input operands will generally cause the DENORM exception (see "Status Flags" above) and correctly signed ZERO results. FAST mode suppresses the DENORM exception. In either FAST or IEEE, DNRM•ZERO will be ZERO without exception. DNRM•INF will be a correctly signed INF without exception in IEEE mode and a NAN and INVALOP in FAST mode. DNRM•NAN will be a correctly signed NAN with INVALOP asserted. The sign bit of the NAN generated from any invalid operation will depend on the operands. (The IEEE Standard does not specify conditions for the sign bit of a NAN.) On the ADSP-3210/3211 Multipliers, the sign of a NAN result will be the exclusive OR of the signs of the input operands.

The product of INF with anything except ZERO or NAN is a correctly signed INF. INF•ZERO will cause INVALOP and yield a NAN. NAN times anything will also cause INVALOP and yield a NAN.

The ADSP-3220/3221 ALUs, in contrast to the Multipliers, are instruction driven with the operation specified by I_{8-0}. The ALU instructions fall into four categories: Fixed-Point, Logical, Single-Precision Floating-Point, and Double-Precision Floating-Point. Instructions are summarized in Tables V through VIII and described in this section below. The data format of results and status flags from the various ALU operations are shown in Tables XI and XII. Division is shown in Tables XIII and XIV; square root in Table XV. Conversions are illustrated in Tables XVI, XVII, and XVIII.

The ADSP-3220/3221 Fixed-Point Arithmetic Operations are:

Mnemonic	Instruction (I_{8-0})			Description
	I_{8-6}	I_{5-3}	I_{2-0}	
IADD	001	000	011	Fixed-point A + B
ISUBB	001	001	011	Fixed-point A − B
ISUBA	001	000	111	Fixed-point B − A
IADDWC	001	010	011	Fixed-point A + B with carry
ISUBWBB	001	011	011	Fixed-point A − B with borrow
ISUBWBA	001	010	111	Fixed-point B − A with borrow
INEGA	001	000	101	Fixed-point − A. ABSA/B must be LO.
INEGB	001	001	010	Fixed-point − B. ABSA/B must be LO.
IADDAS	001	100	011	Fixed-point |A + B|
ISUBBAS	001	101	011	Fixed-point |A − B| ABSA/B must be LO.
ISUBAAS	001	100	111	Fixed-point |B − A| ABSA/B must be LO.

Table V. ADSP-3220/3221 Fixed-Point ALU Operations

The ADSP-3220/3221 Logical Operations are:

Mnemonic	Instruction (I_{8-0})			Description
	I_{8-6}	I_{5-3}	I_{2-0}	
COMPL.A	000	000	101	Ones-complement A
COMPL.B	000	001	010	Ones-complement B
PASSA	000	000	001	Pass A unmodified. Set no flags.
PASSB	000	000	010	Pass B unmodified. Set no flags.
AANDB	000	010	010	Bitwise logical AND
AORB	000	100	010	Bitwise logical OR
AXORB	000	110	010	Bitwise logical XOR
NOP	000	000	000	No operation. Preserve status flags. Preserve Output Register contents with ADSP-3221 only.
CLR	100	000	000	Clear all status flags. Data register contents are unaffected.

Table VI. ADSP-3220 3221 ALU Logical Operations

The ADSP-3220 3221 Single-Precision Floating-Point Operations are:

Mnemonic	Instruction (I_{8-0})			Description
	I_{8-6}	I_{5-3}	I_{2-0}	
SADD	111	000	011	SP FltgPt (A + B)
SSUBB	111	000	111	SP FltgPt (A − B)
SSUBA	111	001	011	SP FltgPt (B − A)
SCOMP	111	001	111	SP FltgPt comparison of A to B. Result is (A − B) Greater Than = (OVRFLO HI) Equal = (OVRFLO LO & UNDFLO LO) Less Than = (UNDFLO HI) Unordered = INVALOP HI
SADDAS	011	000	011	SP FltgPt |A + B|
SSUBBAS	011	000	111	SP FltgPt |A − B|
SSUBAAS	011	001	011	SP FltgPt |B − A|
SFIXA	011	001	101	Convert SP FltgPt A to twos-complement Integer
SFIXB	011	001	110	Convert SP FltgPt B to twos-complement Integer
SFLOATA,	011	100	101	Convert twos-complement integer A to SP FltgPt
SFLOATB	011	100	110	Convert twos-complement integer B to SP FltgPt
DOUBLEA	011	101	101	Convert SP FltgPt A to DP FltgPt
DOUBLEB	011	101	110	Convert SP FltgPt B to DP FltgPt
SPASSA	011	110	001	Pass SP FltgPt A. NANs cause INVALOP.
SPASSB	011	110	010	Pass SP FltgPt B. NANs cause INVALOP.
SWRAPA	011	100	001	Wrap SP DNRM A to SP WNRM
SWRAPB	011	100	010	Wrap SP DNRM B to SP WNRM
SUNWRAPA	011	010	001	Unwrap SP WNRM A to SP DNRM
SUNWRAPB	011	010	010	Unwrap SP WNRM B to SP DNRM
SSIGN	011	111	101	Copy sign from SP FltgPt B to SP FltgPt A. Result is [sign B, exponent A, fraction A].
SXSUB	011	111	001	Subtract B exponent from A exponent. Result is [sign A, (expt A − expt B), fraction A] for all data types. If the unbiased exponent ≥ + 128, INF results. If the unbiased exponent is ≤ − 127, ZERO results.
SITRN	011	010	101	Downshift SP FltgPt A mantissa (with hidden bit) logically by the unbiased SP FltgPt B exponent to a 32-bit unsigned-magnitude integer. Use RZ only.
ADSP-3221 ALU only:				
SDIV	011	110	111	SP FltgPt (A ÷ B)
SSQR	111	110	110	SP FltgPt $\sqrt{B}$

Table VII. ADSP-3220 3221 ALU Single-Precision Floating-Point Operations